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

Anonymous Author(s)

Appendices

A 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?
RQ2: How pythonic idioms are used in real projects?
RQ3: Where are conciseness of pythonic idioms manifested?
RQ4: What are the negative effects of using pythonic idioms?

A.1 RQ1: Challenges in Understanding Pythonic Idiom Syntax and Semantics

A.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.

A.1.2 Approach. We focus on the nine pythonic idioms identified in [18], including list/set/dict-comprehension, chain-comparison, truth-value-test, loop-else, assign-multi-targets, for-multi-targets and star. 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. Then we determine whether the returned top-30 questions for each pythonic idiom are related to the challenges of reading and understanding the concerned idiom. Finally, we collect 154 questions for nine pythonic idioms for analysis.

A.1.3 Result. Our analysis of these idiom-related Stack Overflow questions reveals that Python users are often confused with and even misunderstand unusual syntax and subtle semantics of pythonic idioms. Table 1 presents representative examples for each idiom. (1) or (2) in #R identifies the challenge in these examples. We excerpt and highlight relevant content in red. The details are as follows:

(1) Python users are often confused with the unusual syntax of pythonic idioms. Among the collected questions, we find that Python users may not know the existence of certain pythonic idioms, or they do not understand what the idioms mean even although they know certain idioms are available. It occurs in all nine pythonic idioms and accounts for 63.0% (97 of 154 questions). For example, for the star of the last row of Table 1, since a developer did not know the star idiom, he/she asked if there is a way to expand a Python tuple into a function as actual parameters. Although it was asked before 13 years, it was still active within 1 year and was viewed more than 314,000 times. It indicates many Python users may encounter similar problems. As another example, for the list-comprehension of the 1st row of 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.

Table 1: Challenges in understanding pythonic idioms

Idiom	#R	Question
List Comprehension	(1)	**Question:** Explanation of how nested list comprehension works? I have no problem understanding this: b = [x for x in a] ..., but then I found this snippet: b = [x for xs in a for x in xs] The problem is I'm having trouble understanding the syntax in [x for xs in a for x in xs], could anyone explain how it works? **Asked** 9 years ago; **Modified** 10 months ago; **Viewed** 28k times
Set Comprehension	(2)	**Question:** Set comprehension gives "unhashable type" (set of list) in Python I want to collect all second elements of each tuple into a set: my_set.add({tup[1] for tup in list_of_tuples}) But it throws the following error: TypeError: unhashable type: 'set' **Asked** 5 years ago; **Modified** 5 years ago; **Viewed** 8k times
Dict Comprehension	(1)	**Question:** Can I use list comprehension syntax to create a dictionary? For example, by iterating over pairs of keys and values: d = {... for k, v in zip(keys, values)} **Asked** 13 years ago; **Modified** 9 months ago; **Viewed** 1.1m times
Chain Comparison	(2)	**Question:** Why does 1 in [1,0] == True evaluate to False? ... There seemed to be two ways (to my mind) of evaluating the expression. But with my example 1 in ([1,0] == True) doesn't make sense as an expression, ..., only one seems possible: »> (1 in [1,0]) == True **Asked** 10 years ago; **Modified** 2 years ago; **Viewed** 6k times
Truth Value Test	(2)	**Question:** if x:, vs if x == True, vs if x is True However, I have now discovered that x does not need to be literally "True" for the function to run. So any value other than False appears to evaluate to True, which would not be the case for if x == True or if x is True. **Asked** 9 years ago; **Modified** 5 months ago; **Viewed** 62k times
Loop Else	(1)	**Question:** Why does python use 'else' after for and while loops? **Asked** 10 years ago; **Modified** 2 months ago; **Viewed** 282k times
Assign Multiple Targets	(2)	**Question:** How do chained assignments work? A quote from something: x = y = somefunction() is the same as y = somefunction(); x = y; Is x = y = somefunction() the same as x = somefunction(); y = somefunction()? Based on my understanding, they should be same. **Asked** 11 years ago; **Modified** 9 months ago; **Viewed** 20k times
For Multiple Targets	(1)	**Question:** Two iterating variables in a for loop This is highly confusing to me because I just got my head wrapped around the basic concept of for loops. **Asked** 7 years ago; **Modified** 7 years ago; **Viewed** 20k times
Star	(1)	**Question:** Is there a way to expand a Python tuple into a function as actual parameters? **Asked** 13 years ago; **Modified** 1 year ago; **Viewed** 314k times

(2) Python users often misunderstand the subtle semantics of pythonic idioms. We find Python users can misunderstand the meaning of the idiom which cause unexpected behaviors, or they think that the use of pythonic idioms causes unexpected behaviors, which is applicable to all nine pythonic idioms and accounts for 37.0% (57 of 154 questions). For example, for the assign-multi-targets example 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 has 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 version of non-idiomatic code are different. If the somefunction() is mutable, the first version assigns x and y different values, but the second version assigns the same value to x and y. As another example (see 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 throws the unhashable type error because

items of a set in Python are immutable so 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.

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.

A.2 RQ2: Pythonic Idiom Usage in the Wild

A.2.1 Motivation. The python community make efforts to design versatile pythonic idioms so that the idioms can be used in a variety of ways [4]. Such usage variants have impact on readability and learning difficulty [3, 15]. Exploring coding practices with pythonic idioms in real projects help us understand the typical and unusual (if any) usage scenarios of pythonic idioms.

A.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. We examine the usage of the nine unique pythonic idioms identified in Zhang et al. [18] in these repositories. For the star idiom, Zhang et al. [18] 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 **??**) to identify the idiomatic code instances for the nine idioms. Our approach identifies the 1,708,831 idiomatic code instances for the nine pythonic idioms in the 7,577 repositories. Table 2 shows the statistics of repositories, files and idiomatic code instances for 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 use at least one pythonic idiom, which accounts for 92.3%. Of the 6,997 repositories, 222,637 files and 1,708,831 idiomatic code instances use at least one pythonic idiom. We observe that variant usage of pythonic idioms is manifested in three aspects: diversity of usage context of pythonic idioms, diversity of data objects used by pythonic idioms, and the variety of AST node components of pythonic idioms.

A.2.3 Diversity of usage context of pythonic idioms. An idiom is like a phrase. To understand an idiom we should relate it to its context [10–13, 16]. A pythonic idiom corresponds to an AST node, it is usually attached as a child of another AST node. We extract the parent node of each idiomatic code fragment as the usage context. Three pythonic idioms (i.e., loop-else, assign-multi-targets and for-multi-targets) are of statement type. As statements are units of code that Python interpreter can execute, these three idioms can be used as an instruction to perform a complete operation (e.g, assign values to data objects) [5, 7–9] in any context. Therefore, we analyze only the remaining six pythonic idioms.

Figure 1 shows the usage context of six pythonic idioms (i.e., list/set/dict-comprehension, chain-comparison, truth-value-test and star). Each bar means the percentages of each usage context among all usage contexts of each pythonic idiom. We linear-scale up the bar for the percentage less than 10%.

The list/set/dict-comprehension all have more than 20 types of parent nodes (i.e., more than 20 different usage context). All of them are most frequently used as values of Assignment statements (56%-69%), and are also often used as arguments of Function Calls (10%-25%) and values of Return statements (8%-13%). These three usage contexts together account for over 80% for the list-, set-, and dict-comprehension, respectively. The remaining usage contexts of list/set/dict-comprehension are unusual. For example, Figure **??** shows the list-comprehension is used as the body of Lambda node to create a data object, which accounts for about 0.2% of all usage contexts. For the chain-comparison, there are 14 kinds of parent nodes. It is usually used as the condition of If (47%) and Assert (38%) statements. For the truth-value-test and star, the types of parent nodes are 4 (If, IfExp, Assert, While) and 5 (Call, Tuple, List, Set, ClassDef), respectively. The truth-value-test is most frequently used as the condition of If statements (84%), and the star is most frequently used as arguments of Function Calls (94%).

A.2.4 Diversity of data objects that pythonic idioms use. Pythonic idioms may use different kinds of data, The more diverse the data is, the harder it is to read and understand pythonic idiom. The data objects of list/set/dict-comprehension, chain-comparison, truth-value-test, star, loop-else, assign-multi-targets and for-multi-targets correspond to added element, chained comparator, testing object, starred value, iterated object, value and target, the iterated object, respectively. We calculate the frequency of the AST node type corresponding to these data objects used by pythonic idioms.

Figure 2 shows the distribution of the percentages of AST node types of data objects used by nine pythonic idioms. There are 25

Table 2: The statistics of repositories, files and idiomatic code instances of nine pythonic idioms

Idiom	Repos	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

Table 3: The number of AST node components of pythonic idioms

Idiom	Node	Max	Min	Med	Code	% (>Med)	Repos	% (>Med)
List-Compr	For	7	1	1	4746	1.5	1230	20.5
	If	9	0	0	54504	17.4	4127	68.7
	IfElse	16	0	0	8397	2.7	1695	28.2
Set-Compr	For	4	1	1	551	6.0	183	17.7
	If	2	0	0	2270	24.9	536	51.7
	IfElse	2	0	0	91	1.0	59	5.7
Dict-Compr	For	4	1	1	489	1.2	222	7.1
	If	3	0	0	7342	18.4	1553	50.0
	IfElse	4	0	0	1382	3.5	417	13.4
Chain Compare	Cmpop	10	2	2	1202	4.9	384	14.7
Loop-Else	Break	10	0	1	538	9.5	282	21.1
Ass Multi Tar	=	28	1	1	21458	4.1	2421	38.4
	Target	65	2	2	150327	28.6	4891	77.5
	Starred	2	0	0	1962	0.4	503	8.0
For-Multi Tar	Starred	2	0	0	183	0.1	85	1.4
	Target	46	2	2	30556	13.8	3484	58.4

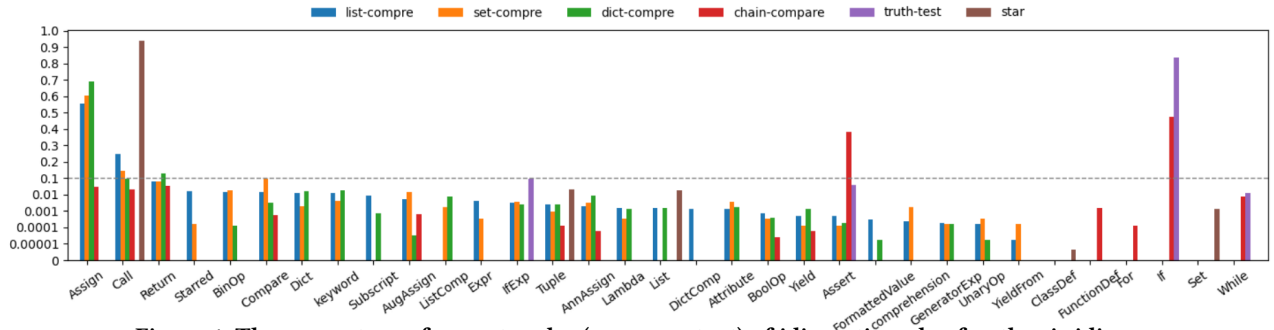


Figure 1: The percentage of parent nodes (usage context) of idiomatic code of pythonic idioms

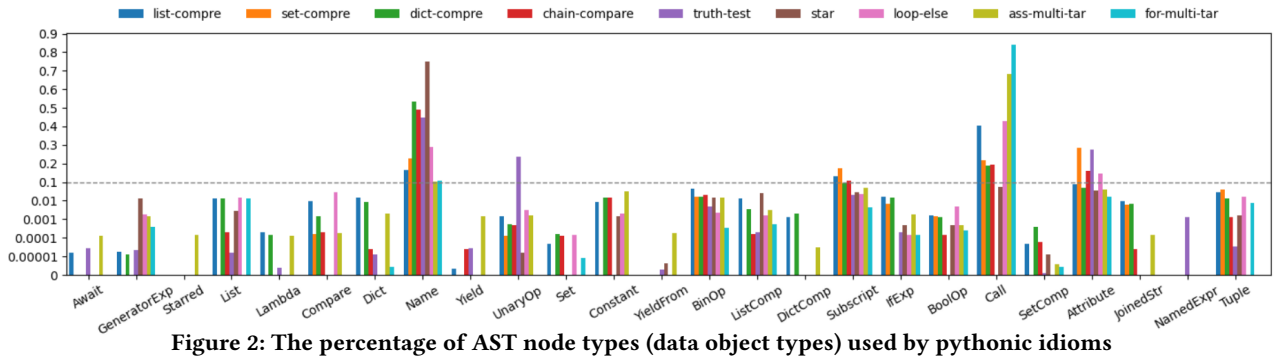


Figure 2: The percentage of AST node types (data object types) used by pythonic idioms

types of AST nodes used by nine pythonic idioms, and the types of AST nodes used by each idiom all exceed 10. Among 25 AST node types, four AST node types are commonly used by nine pythonic idioms: Name, Function Call, Attribute Access and Subscript. The total percentage of these four AST node types for each idiom is all over 70%. The remaining 21 types of AST node are infrequently used data objects. For examples, for the star idiom, the data object of *'mp4v' is a Constant node. For the code "for (prev_node, channel) in node.outputs if rev else node.inputs" of for-multi-targets, the iterated object is an IfExpr node.

A.2.5 Distribution of AST node components of pythonic idioms. Pythonic idioms may consist of different kinds of nodes, e.g, list/set-/dict-comprehension may consist of three kinds of nodes: For, If and If-Else. For each idiom, the more its node components are, the more difficult it is to read and use the idiom correctly and quickly [3, 15]. As truth-value-test and star only have one node component, we do not consider these two idioms. For the other seven idioms, we compute the number of occurrences of each node component for all idiomatic code instances of each idiom.

Table 3 shows the statistics of the number of node components of pythonic idioms. The *Code* and *%(>Med)* columns represent how many idiomatic code instances and what percentage of these code instances use more than the median number of node components, respectively. The *Repos* and *%(>Med)* columns represent how many repositories with code instances of a pythonic idiom and what percentage of these repositories use more than the median number of node components, respectively. For five idioms (list-comprehension, chain-comparison, assign-multi-targets, loop-else and for-multi-targets), each idiom has at least one node component with Max>10.

Idiomatic code:

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

Non-idiomatic code:

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

Figure 3: The idiomatic code and the corresponding non-idiomatic code of loop-else

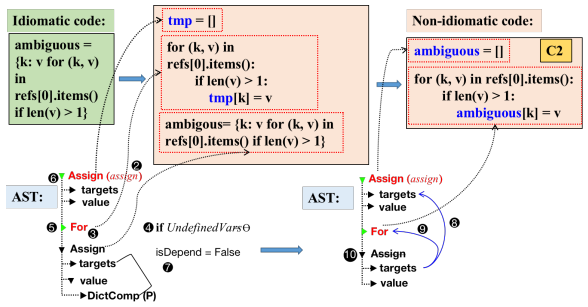
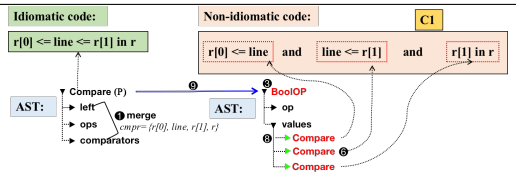
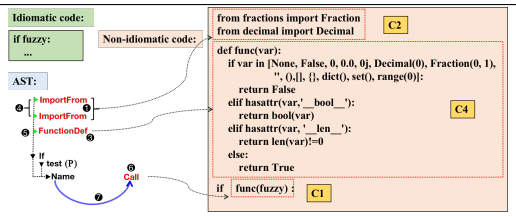
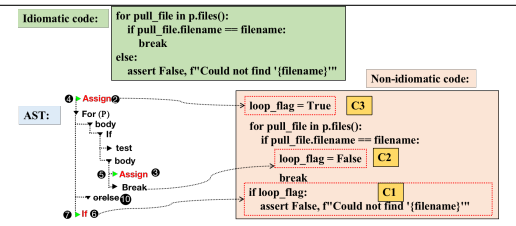
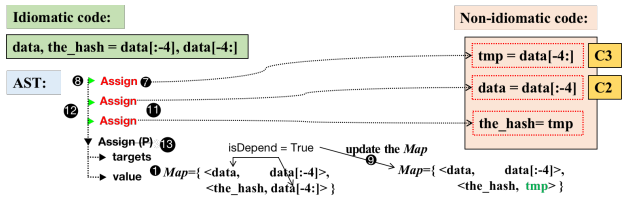
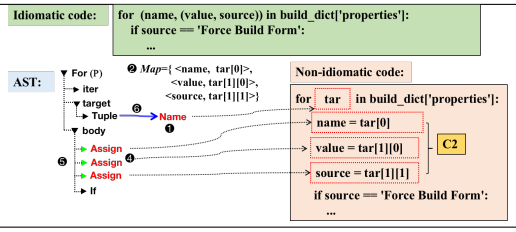
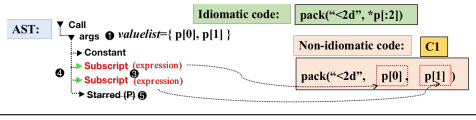
The maximum of targets of assign-multi-targets and for-multi-targets are greater than 40. For set/dict-comprehension, each idiom has at least one node component with a maximum value reaches 4. Although the median values of node components of all idioms are less than 3, the percentages of idiomatic code instances containing more than median node components cannot be overlooked, which can reach up to 28.2% and 68.7% (the *%(>Med)* column of idiomatic code instances and repositories), respectively. For example, for list-comprehension, among its three AST node components, the If node has the highest *%(>Med)* of *Code* (17.4%) and *Repos* (68.7%).

Pythonic idioms may be used in diverse contexts, use diverse data objects, and contain complex node components. In addition to some typical usage, there are many long-tail usage scenarios due to the community commitment to make pythonic idioms versatile. However, such diverse usage scenarios may not be clearly documented or commonly received, which may hinder the understanding of pythonic idioms and the adoption of pythonic idioms in broader practice.

A.3 RQ3: Concise of Pythonic Idioms

Figure 3 shows an example of loop-else that leads to redundant code. The idiomatic code can be implemented without the else keyword and deindent the body in the else clause. Compared to loop-else, the corresponding non-idiomatic code removes the else: line in the idiomatic code.

Table 4: Rules of refactoring idiomatic code into non-idiomatic code for nine pythonic idioms

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

P represents idiomatic code of Python idioms; isDepend(n_1, n_2) represents whether there is data dependence between n_1 and n_2 nodes; pos(n) represent the position of n in the abstract syntax tree.

Idiomatic code | Non-idiomatic code | Create a node | Insert a node to somewhere | Replace | XXXX Remove a node | Concise manifestation

B APPROACH

Table 4 shows the details of refactoring idiomatic code into non-idiomatic code of nine pythonic idioms.

C EVALUATION

To evaluate our approach, we study two research questions:

RQ1 (Accuracy): How accurate is our approach when refactoring Python idiomatic code into non-idiomatic code?

RQ2 (Usefulness): Is the generated non-idiomatic code useful for explaining and understanding pythonic idiom usage and errors?

C.1 RQ1: Refactoring Accuracy

Testing based verification We statically collect test cases to evaluate the correctness of the code refactoring, because dynamically running all test cases of a project before and after code refactoring for 1,708,831 idiomatic code instances is not realistic and cannot add value compared to statically collecting test cases. We use DLocator [17] statically analyze code to collect test cases that directly call the methods of the idiomatic code. After that, we execute the test cases before and after refactoring the idiomatic code. If the test cases pass in both cases, the code refactoring is correct. Otherwise, if the test cases pass before code refactoring but fail after refactoring, we think the code refactoring is wrong and manually analyze the reason for the failure is because of the detection or the refactoring of idiomatic code. We use Pytest [14] to run test cases because Pytest is a widely used unit testing frame in Python projects and supports the Python's default unittest tool. The complete process is shown in Figure 4.

C.2 RQ2: Refactoring Usefulness

C.2.1 Performance Comparison and Analysis. Table 5 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). For the answer correctness, G1 and G2 achieve 0.41~0.81 and 0.87~1 for nine pythonic idioms, respectively. The improvement of correctness is 23.5%~136.6% for different idioms. For all 27 questions, the overall correctness of G1 and G2 is 0.54 and 0.94, respectively. The improvement is 74.1% overall. 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 338 seconds (about 6 minutes) to 2025 seconds (about 33 minutes) among the 20 participants. For each pythonic idiom, only for the assign-multi-targets idiom where G2 takes about 7.5% 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 2.3%~32.7% less time than G1, even they have to read both idiomatic and non-idiomatic code. This suggests that the generated non-idiomatic code can speed up the understanding of pythonic idioms.

The detailed discussion about performance comparison and analysis of nine pythonic idioms are as follows:

• **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 (0.58 and 0.65) respectively. According to Zhang et al. [18], list-comprehension is much more frequently used than dict/set-comprehension. Our prior idiom knowledge survey in Figure 5 also suggests that the participants generally know better and use more frequently list-comprehension than dict/set-comprehension. With the explanatory non-idiomatic code, the correctness gap between the three comprehension idioms becomes very small (1, 0.95 and 0.95 for list/dict/set 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 18.2%.

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 participants in G1 answer wrongly, and the correctness is only 40%. In contrast, only one participant of G2 answers wrongly (90% correctness). This 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 40.6%, with 14.5% speed-up. The truth-value-test involves a variety of situations, e.g, any object like Call, BinOp and Attribute can be directly 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 as listed in the 3rd row of Table 4. Python users generally understand that None and 0 are considered false, but it is hard to grasp all other situations. For example, the value of the idiomatic code if d.expression is Decimal(0). Most of participants of G1 (7 of 10) answer wrongly but no one in G2 answers wrongly. For the idiomatic code, the average completion time of G1 and G2 is 42.5s and 36.3s, respectively. The non-idiomatic code makes the G2 participants spend 14.6% less time than G1. As the corresponding non-idiomatic code contains two explicit statements (see the 3rd row of Table 4): "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.

Table 5: Performance Comparison

Idioms	Num	Correctness			Time (s)		
		G1	G2	Imprv (%)	G1	G2	Imprv (%)
List-Comprehension	3	0.81	1	23.5	47.2	38.6	18.2
Set-Comprehension	3	0.65	0.95	46.1	50.7	34.6	31.8
Dict-Comprehension	3	0.58	0.95	63.8	78.8	67.9	13.8
Chain-Comparison	3	0.41	0.9	119.5	41.3	27.8	32.7
Truth-Value-Test	3	0.69	0.97	40.6	33.9	29	14.5
Loop-Else	3	0.41	0.97	136.6	59.2	41.3	30.2
For-Multi-Targets	3	0.62	0.87	40.3	48.2	47.1	2.3
Assign-Multi-Targets	3	0.46	0.97	110.9	22.8	24.5	-7.5
Star	3	0.62	0.9	45.2	45.5	30.8	32.3
All	27	0.54	0.94	74.1	46.09	36.8	20.2

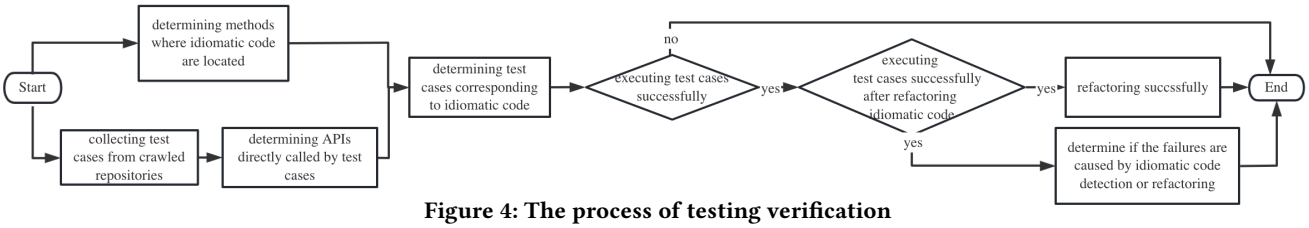


Figure 4: The process of testing verification

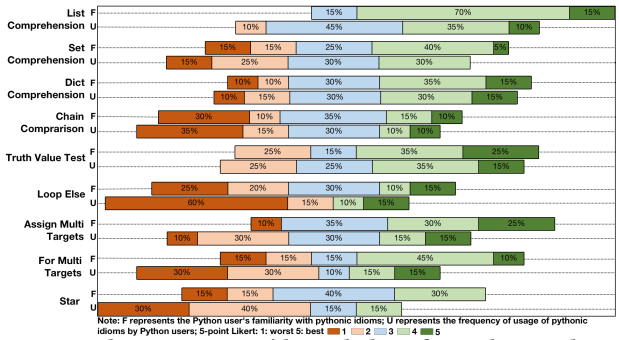


Figure 5: The participants' knowledge of 9 pythonic idioms

• **chain-comparison idiom:** The improvement of correctness is 119.5%, with 32.7% speed-up. We find many Python users understand >=, <=, >, = and < chained operators correctly. However, when it involves is, is not, in and not in operators, they generally cannot understand correctly. It is because they wrongly assume that these operators have priority order or wrongly explain comparison operators from left to right. However, all comparison operations in Python have the same priority. The chain-comparison is semantically equal to union of several comparison operations. It echos well the negative effects caused by the chain-comparison idiom in Section A.2. Non-idiomatic code by our tool can effectively clarify those common misunderstandings and thus result in faster and more correct understanding of chain comparisons.

• **loop-else idiom:** The improvement of correctness is 136.6%, with 30.2% speed-up. Most of Python users (6 of 10 participants in G1) answer that they do not understand the loop-else idiom or they think the loop-else syntax is wrong. We interview them and summarize two reasons: they assume that the else clause can only be added after the if statement or they cannot guess what the else clause does in the code. We find that the explanatory non-idiomatic code can not only help participants realize Python supports the else clause after for and while statements but also help them correctly understand the meaning of loop-else.

• **for-multi-targets idiom:** G2 has relatively lower correctness score (0.87), but it is still much higher than 0.62 for G1 on for-multi-targets. Furthermore, G2 has marginal understanding speed-up (only 2.3%). We interview participants in G2 and find participants in G2 need to read more assignment statements, and values of these assignment statements are Subscript AST nodes (i.e., element access). As the number of targets and the nesting depth increases, non-idiomatic code has more assignment statements and each value of the assignments has more Subscript nodes than idiomatic code. For example, for the idiomatic code "for (i, j), t_ij in single_amplitudes", the corresponding non-idiomatic code explains the "j" as "j = tar[0][1]" in the body of the "for tar in single_amplitudes". The for statement has three targets, the non-idiomatic code has three more assignment statements and "i" and "j" have two Subscript nodes, which makes G2 participants sometimes accidentally misread the code and spend more time.

• **assign-multi-targets idiom:** The improvement of correctness is 110.9%, but with 7.5% slow-down in the understanding time. One common misconception G1 participants have for assign-multi-targets is the evaluation order of targets. For example, for the code "dummy2 = other = ListNode(0)", all G1 participants assume that "other" is assigned first, but "dummy2" is assigned first. The explanatory non-idiomatic code "tmp=ListNode(0); dummy2 = tmp; other = tmp" makes G2 participants avoid this misunderstanding. The other common misconception is the evaluation order of targets and value. For example, for the code "uc, ud = ud - q * uc, uc", 5 G1 participants think it is equal to "uc = ud - q * uc; ud = uc". It is because they do not realize the right-hand side (value of assign-multi-targets) is always evaluated before the left-hand side (targets of the assign-multi-targets). The explanatory non-idiomatic code "tmp = uc; uc = ud - q * uc; ud = tmp" helps G2 participants realize the correct evaluation order. These evaluation order misconceptions are also reflected in some Stack Overflow questions we investigated [1, 2].

• **star idiom:** The improvement of correctness is 45.2%, with 32.3% speed-up. Star can operate any iterable objects such as Subscript and Constant, and it can be used as parameters in the function call, the targets of for and targets and values of assignment statements. We find participants in G1 generally can answer question correctly for the * before a Subscript data object, but many G1 participants cannot understand other circumstances correctly. For example, for the idiomatic code "cv2.VideoWriter_fourcc(*'mp4v')", we ask them about the number of arguments passed by the function call. 2 G1 participants think the code has syntax error and 3 G1 participants think the number of arguments is one, so the correctness is 50%. We interview them and find them do not know the star can be applied to the string constant or they think the number of elements to be unpacked is 1. The provided non-idiomatic code "cv2.VideoWriter_fourcc('m','p', '4', 'v')" help Python users avoid such misunderstandings.

D PRACTICAL NOTES FOR PYTHON USERS

Python users could learn pythonic idioms from their usage context, data objects and AST node components supported by pythonic idioms according to Section A.2. For example, list/set/dict-comprehension as an expression type AST node can be a child of almost any statement, e.g., used as the values of Assignment statement and Return statement. Although list/set/dict-comprehension can be used as an

individual statement, we do not recommend such usage because the idioms use more memory and run slower compared to loop statements [6]. For example, a developer wrote a code [CL.remove(m) for m in CL...], then another developer submitted a pull request to remove the usage of the list-comprehension and the request was merged. Besides, Python supports the nesting of multiple for, if and if-else keywords. The if-else keyword is placed in front of the for keyword, indicating that different elements are added to the object under different conditions. For multiple same keywords such as two if-else keywords, two for keywords or two if keywords, Python users should read from left to right.

When Python users encounter pythonic idioms, they need to be careful about the influence of other programming languages. For example, for the chain-comparison, all comparison operators have the same precedence. Our refactoring tool helps reduce confusion and misunderstanding of pythonic idioms. Furthermore, Python users can debug programs by refactoring idiomatic code into non-idiomatic code. For example, in the set-comprehension row of Table 1, the idiomatic code "my_set.add({tup[1] for tup in list_of_tuples})" throws the unhashable type: 'set', because set cannot be added to the "my_set" as an element. If we refactor the idiomatic code into the non-idiomatic code, we need to create a variable "tmp" to save the value of "{tup[1] for tup in list_of_tuples}", and then "my_set.add({tup[1] for tup in list_of_tuples})" is replaced by "my_set.add(tmp)". Reading explanatory non-idiomatic code would help Python user realize the root-cause of the error.

REFERENCES

[1] 2022. *Assign-Multi-Targets.* https://stackoverflow.com/questions/8725673/multiple-assignment-and-evaluation-order-in-python

[2] 2022. *Assign-Multi-Targets.* https://stackoverflow.com/questions/7601823/how-do-chained-assignments-work

[3] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering* 36, 4 (2009), 546–558.

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

[5] Python Developers. 2011. *The difference between Python statements and Python expressions.* https://stackoverflow.com/questions/4728073/what-is-the-difference-between-an-expression-and-a-statement-in-python

[6] Python Developers. 2015. *The performance of the list-comprehension.* https://stackoverflow.com/questions/22108488/are-list-comprehensions-and-functional-functions-faster-than-for-loops

[7] Python Developers. 2022. *Understanding statements in Python.* https://codepict.com/expressions-and-statements-in-python/

[8] Python Developers. 2023. *Python statements and Python expressions.* http://en.wikipedia.org/wiki/Python_(programming_language)#Expressions

[9] Allen Downey. 2012. *Think python.* " O'Reilly Media, Inc.".

[10] Timothy F Doyle. 2007. *The role of context in meaning and understanding.* Ph. D. Dissertation. Universitaet Potsdam.

[11] Michael AK Halliday. 1999. The notion of "context" in language education. *AMSTERDAM STUDIES IN THE THEORY AND HISTORY OF LINGUISTIC SCIENCE SERIES 4* (1999), 1–24.

[12] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering.* IEEE, 232–242.

[13] Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering.* 117–125.

[14] John Hunt. 2019. PyTest Testing Framework. In *Advanced Guide to Python 3 Programming.* Springer, 175–186.

[15] Yahya Tashtoush, Zeinab Odat, Izzat M Alsmadi, and Maryan Yatim. 2013. Impact of programming features on code readability. (2013).

[16] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 38–49.

[17] 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.* 233–244.

[18] 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. *arXiv preprint arXiv:2207.05613* (2022).