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

We summarize two key challenges: unusual syntax and subtle semantics. 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:

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; Active 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

(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 asked if anyone can explain how it works.

(2) Python users often misunderstand the subtle semantics of pythonic idioms. We find Python users can misunderstand the meaning of the idiom 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 153 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

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.

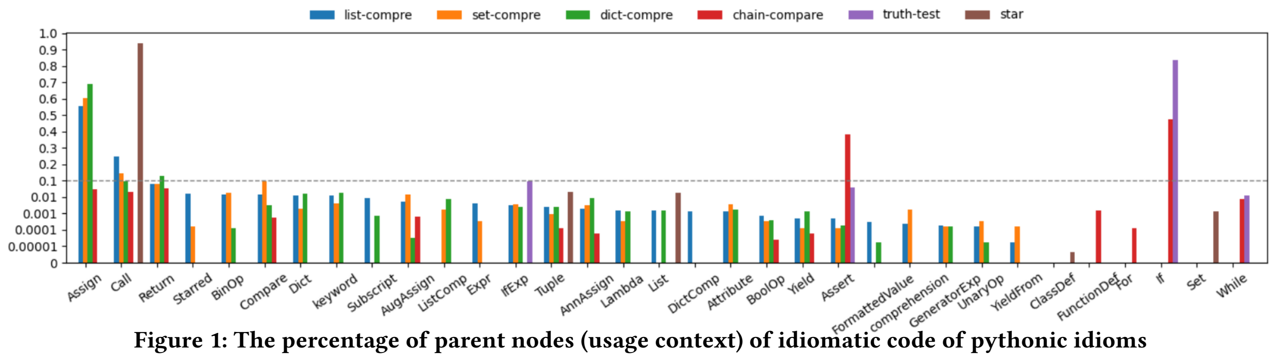


Figure 1: The percentage of parent nodes (usage context) of idiomatic code of pythonic idioms

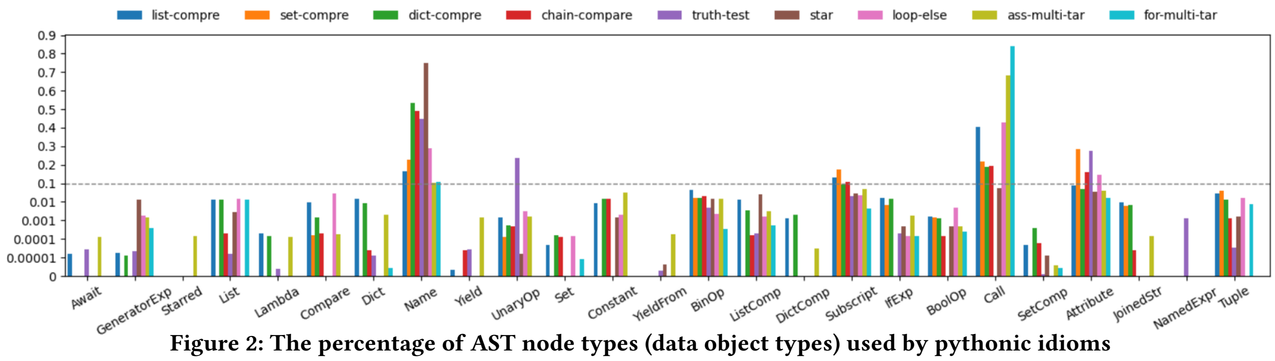


Figure 2: The percentage of AST node types (data object types) used by pythonic idioms

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

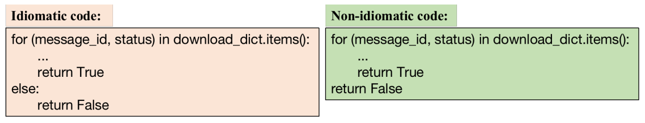
Figure 1 shows the usage context of six pythonic idioms (i.e., list/set/dict-comprehension, chain-comparison, truth-value-test and star). Each bar means the percentages of each usage context among all usage contexts of each pythonic idiom. We scale up the vertical axis which is less than 10%.

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

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.

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


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

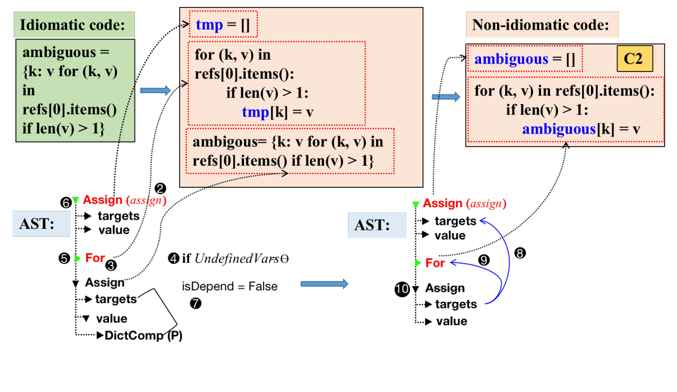
A.3 RQ3: Concise of Pythonic Idioms

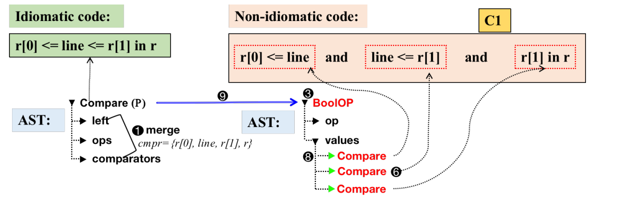
A.4 RQ4: Negative Effects Caused by Idiom Misuse and Misunderstanding

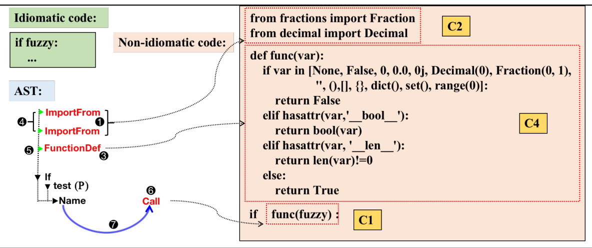
Figure 3 shows an example of loop-else that leads to redundant code. The idiomatic code can be implemented without the else keyword and deindent the body in the else clause.

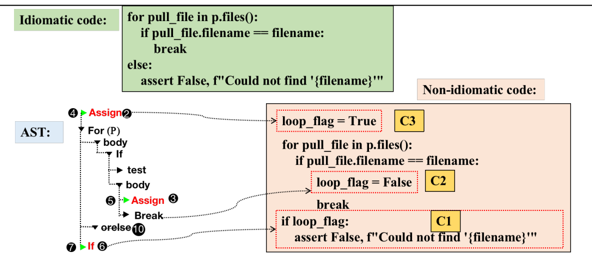
Table 4: Rules of refactoring idiomatic code into non-idiomatic code for nine pythonic idioms

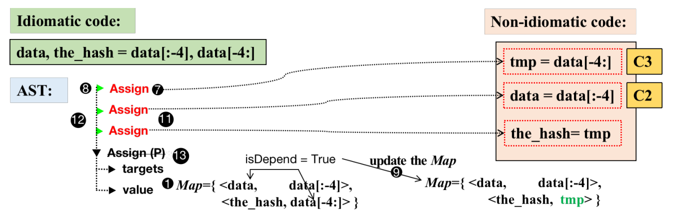
Idiom	Examples of Code Refactoring	Transformation Steps

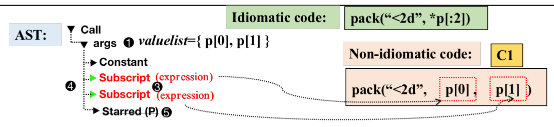
List/ Set/ Dict Compre-hension	 Idiomatic code:
	`ambiguous = {k: v for (k, v) in refs[0].items() if len(v) > 1}`
	Non-idiomatic code:
	`ambiguous = []` C2
	`for (k, v) in refs[0].items():`
	`    if len(v) > 1:`
	`        ambiguous[k] = v`
		1: Get variables $UndefinedVars$ from P that do not appear 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	 Idiomatic code:
	`r[0] <= line <= r[1] in r`
	Non-idiomatic code: C1
	`r[0] <= line and line <= r[1] and r[1] in r`
		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	 Idiomatic code:
	`if fuzzy:`
	`    ...`
	Non-idiomatic code:
	`from fractions import Fraction`
	`from decimal import Decimal` C2
	`def func(var):`
	`    if var in [None, False, 0, 0.0, 0j, Decimal(0), Fraction(0, 1),`
	`            '', (),[], {}, dict(), set(), range(0)]:`
	`        return False`
	`    elif hasattr(var,'__bool__'):`
	`        return bool(var)`
	`    elif hasattr(var, '__len__'):`
	`        return len(var)!=0`
	`    else:`
	`        return True` C4
	`if func(fuzzy) :` C1
		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, $call$)

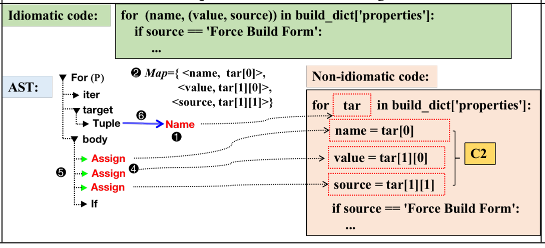
Loop Else	 Idiomatic code:
	`for pull_file in p.files():`
	`    if pull_file.filename == filename:`
	`        break`
	`else:`
	`    assert False, f"Could not find '{filename}'"`
	Non-idiomatic code:
	`loop_flag = True` C3
	`for pull_file in p.files():`
	`    if pull_file.filename == filename:`
	`        loop_flag = False` C2
	`        break`
	`if loop_flag:`
	`    assert False, f"Could not find '{filename}'"` C1
		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	 Idiomatic code:
	`data, the_hash = data[:-4], data[-4:]`
	Non-idiomatic code:
	`tmp = data[-4:]` C3
	`data = data[:-4]` C2
	`the_hash= tmp`
		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)

Star	 Idiomatic code:
	`pack(“<2d”, *p[:2])`
	Non-idiomatic code: C1
	`pack(“<2d”, p[0], p[1])`
		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

Table 5: Continued

Idiom	Examples of Code Refactoring	Transformation Steps
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)

Idiomatic code:

```
for  (name, (value, source)) in build_dict['properties']:
  if source == 'Force Build Form':
    ...
```

Non-idiomatic code:

```
for  tar  in build_dict['properties']:
  name = tar[0]
  value = tar[1][0]
  source = tar[1][1]
  if source == 'Force Build Form':
    ...
```

AST:
For (P)
iter
target
Tuple
Name
body
Assign
Assign
Assign
If
❷ Map={ <name, tar[0]>, <value, tar[1][0]>, <source, tar[1][1]>}
❶
❹
❺
❻
C2

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

Table 5: Performance Comparison

Idioms	Num	Correctness			Time (s)		
		G1	G2	Imprv (%)	G1	G2	Imprv (%)
List-Comprehension	3	0.81	1	23.8	47.2	38.6	18.2
Set-Comprehension	3	0.65	0.95	45.3	50.7	34.6	31.8
Dict-Comprehension	3	0.58	0.95	64.7	78.8	67.9	13.8
Chain-Comparison	3	0.41	0.9	119.3	41.3	27.8	32.7
Truth-Value-Test	3	0.69	0.97	40.1	33.9	29	14.5
Loop-Else	3	0.41	0.97	136.4	59.2	41.3	30.2
For-Multi-Targets	3	0.62	0.87	41.4	48.2	47.1	2.3
Assign-Multi-Targets	3	0.46	0.97	110	22.8	24.5	-7.4
Star	3	0.62	0.9	46.3	45.5	30.8	32.3
All	27	0.54	0.94	74.1	46.09	36.8	20.2

B APPROACH

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

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 1,584,695 idiomatic code instances is not realistic and cannot add value compared to statically collecting test cases. We use DLocator [37] 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. The complete process is shown in Figure 4. Particularly, we use Pytest [22] to run test cases because Pytest not only is a widely used unit testing frame in Python projects but also supports the Python's default unittest tool.

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). 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. [41], 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 40.1%, 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

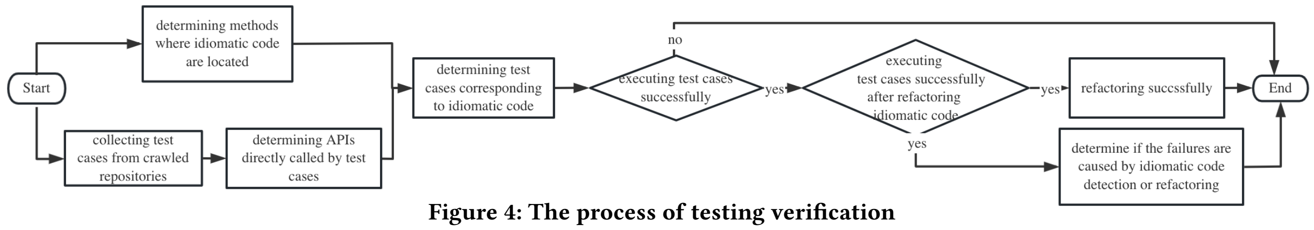

Start
determining methods where idiomatic code are located
collecting test cases from crawled repositories
determining APIs directly called by test cases
determining test cases corresponding to idiomatic code
executing test cases successfully
no
yes
executing test cases successfully after refactoring idiomatic code
yes
yes
refactoring succssfully
determine if the failures are caused by idiomatic code detection or refactoring
End

Figure 4: The process of testing verification

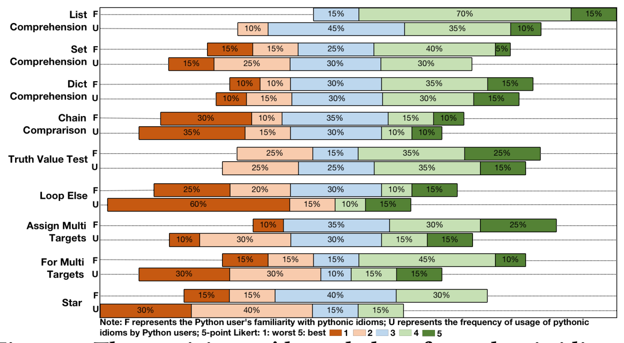


Figure 5: The participants' knowledge of 9 pythonic idioms

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.

• **chain-comparison idiom:** The improvement of correctness is 119.3%, 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 136.4%, 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%, but with 7.4% 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 46.3%, 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.

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] 2022. *The Explanation of Chain Comparison*. https://stackoverflow.com/questions/101268/hidden-features-of-python
[4] 2022. *The Explanation of Chain Comparison*. https://stackoverflow.com/questions/58084423/strange-chained-comparison
[5] 2022. *Pylint*. https://pylint.readthedocs.io/en/latest/
[6] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th european conference on software maintenance and reengineering*. IEEE, 181–190.
[7] Carol V Alexandru, José J Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C Gall, and Gregorio Robles. 2018. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 1–11.
[8] Arooj Arif and Zeeshan Ali Rana. 2020. Refactoring of code to remove technical debt and reduce maintenance effort. In *2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*. IEEE, 1–7.
[9] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software*. IEEE, 23–31.
[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2000. Finding refactorings via change metrics. *ACM SIGPLAN Notices* 35, 10 (2000), 166–177.
[11] Python developers. 2000. *Python Enhancement Proposals*. https://peps.python.org/pep-0000/
[12] Python Developers. 2015. *The performance of the list-comprehension*. https://stackoverflow.com/questions/22108488/are-list-comprehensions-and-functional-functions-faster-than-for-loops
[13] Python Developers. 2018. *The performance of chain comparison*. https://stackoverflow.com/questions/48375753/why-are-chained-operator-expressions-slower-than-their-expanded-equivalent
[14] Python Developers. 2022. *The definition of logical lines of Python program*. ttps://docs.python.org/3/reference/lexical_analysis.html#logical-lines
[15] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *International Conference on Software Engineering (ICSE'22). To appear.*
[16] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. 2006. Does god class decomposition affect comprehensibility?. In *IASTED Conf. on software engineering*. 346–355.
[17] Aamir Farooq and Vadim Zaytsev. 2021. There is More than One Way to Zen Your Python. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. 68–82.
[18] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 204–215.
[19] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[20] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of android code smells. In *Proceedings of the international conference on mobile software engineering and systems*. 59–69.
[21] Raymond Hettinger. 2013. *Transforming code into beautiful, idiomatic Python*. https://www.youtube.com/watch?v=OSGv2VnC0go
[22] John Hunt. 2019. PyTest Testing Framework. In *Advanced Guide to Python 3 Programming*. Springer, 175–186.
[23] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *44th International Conference on Software Engineering (ICSE'22)(Pittsburgh, United States)(ICSE'22). ACM. https://doi. org/10.1145/3510003.3510115*.
[24] Mirko Köhler and Guido Salvaneschi. 2019. Automated refactoring to reactive programming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 835–846.
[25] Pattara Leelaprute, Bodin Chinthanet, Supatsara Wattanakriengkrai, Raula Gaikovina Kula, Pongchai Jaisri, and Takashi Ishio. 2022. Does Coding in Pythonic Zen Peak Performance? Preliminary Experiments of Nine Pythonic Idioms at Scale. *arXiv preprint arXiv:2203.14484* (2022).
[26] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. 2011. An exploratory study of code smells in evolving aspect-oriented systems. In *Proceedings of the tenth international conference on Aspect-oriented software development*. 203–214.
[27] Alex Martelli, Anna Ravenscroft, and David Ascher. 2005. *Python cookbook*. " O'Reilly Media, Inc.".
[28] José Javier Merchante and Gregorio Robles. 2017. From Python to Pythonic: Searching for Python idioms in GitHub. In *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution*. 1–3.
[29] Christian D Newman, Brian Bartman, Michael L Collard, and Jonathan I Maletic. 2017. Simplifying the construction of source code transformations via automatic syntactic restructurings. *Journal of Software: Evolution and Process* 29, 4 (2017), e1831.
[30] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 1–53.
[31] Purit Phan-udom, Naruedon Wattanakul, Tattiya Sakulniwat, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Morakot Choetkiertikul, and Raula Gaikovina Kula. 2020. Teddy: Automatic Recommendation of Pythonic Idiom Usage For Pull-Based Software Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 806–809.
[32] Tattiya Sakulniwat, Raula Gaikovina Kula, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Dong Wang, Takashi Ishio, and Kenichi Matsumoto. 2019. Visualizing the Usage of Pythonic Idioms over Time: A Case Study of the with open Idiom. In *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 43–435.
[33] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.
[34] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[35] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An empirical study of refactorings and technical debt in Machine Learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 238–250.
[36] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 483–494.
[37] 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.
[38] Peter Weißgerber and Stephan Diehl. 2006. Identifying refactorings from source-code changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 231–240.
[39] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring detection based on umldiff change-facts queries. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 263–274.
[40] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346.
[41] 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).