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

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

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

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

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

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

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

Appendices

A APPROACH

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

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

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

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

• **assign-multi-targets:** The assign-multi-targets idiom can assign values to multiple targets in an assignment statement. We detect the idiomatic code that is an Assign node with at least two targets, e.g., the P of the third last row of Table 2 has two targets: data and the_hash. To refactor the idiomatic code into non-idiomatic code, we first map the elements in targets and value of P to get a 2-tuple *Map* (line 1). If there is data dependence between the $i-th$ target and the value after $i-th$, we create a temporary variable to save the value (line 5 and 7). And then we insert the assignment statement to the position of the P and update the *Map* with the temporary

Table 1: Detection rules of the code of nine pythonic idioms

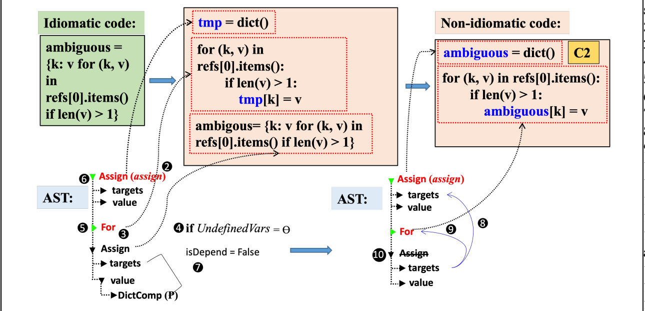
Idiom	Detection Rules
List/Set/Dict Comprehension	P=ListComp/SetComp/DictComp
Chain Comparison	P=Compare and Num(P.ops)>1
Truth Value Test	P.kind=test and P ∉ {Compare, Call, BoolOp}
Loop Else	P ∈ {For, While} and P.orelse ≠ ∅
Assign Multi Targets	P=Assign and Num(P.targets)>1
For Multi Targets	P=For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

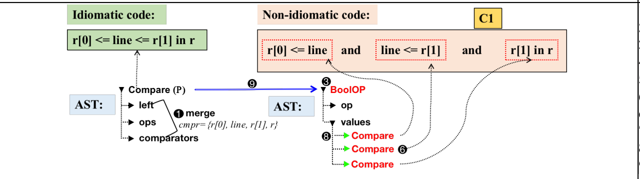
Note: P represents idiomatic code of Python idioms; Num(s) returns the number of elements in s. Other symbols starting with a capital letter indicates AST node types or AST node properties defined in Python language specification.

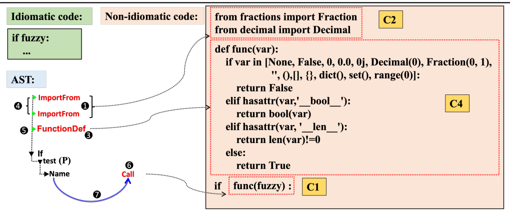
*Corresponding author.

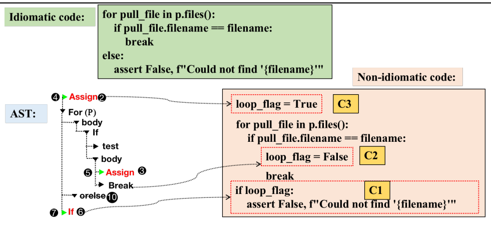
Table 2: Rules of refactoring idiomatic code into non-idiomatic code for nine pythonic idioms

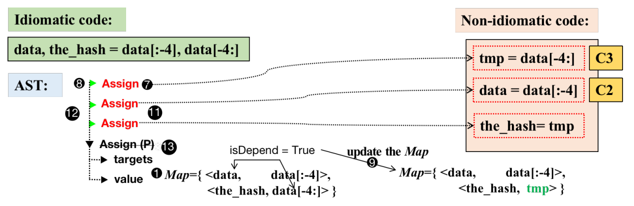
Idiom	Examples of Code Refactoring	Transformation Steps

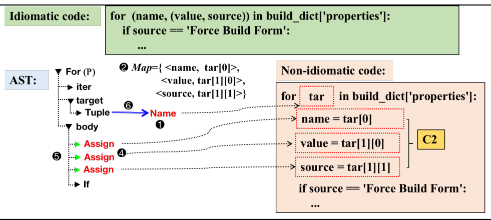
List/ Set/ Dict Compre- hension	 Idiomatic code:
	`ambiguous = {k: v for (k, v) in refs[0].items() if len(v) > 1}`
	`tmp = dict()`
	`for (k, v) in refs[0].items():`
	`    if len(v) > 1:`
	`        tmp[k] = v`
	`ambigous= {k: v for (k, v) in refs[0].items() if len(v) > 1}`
	Non-idiomatic code: C2
	`ambiguous = dict()`
	`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, $call$)

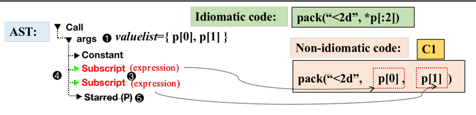
Chain Compa- rison	 Idiomatic code:
	`r[0] <= line <= r[1] in r`
	Non-idiomatic code: C1
	`r[0] <= line and line <= r[1] and r[1] in r`
		1: Merge P.left and P.comparators into $cmpr$
		2: ops=P.ops
		3: $boolnode$=Create("Bool", op="And")
		4: ind = 0
		5: **For** op in ops **do**
		6: $comparenode$=Create("Compare", 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` C2
	`from decimal import Decimal`
	`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__'):` C4
	`        return bool(var)`
	`    elif hasattr(var, '__len__'):`
	`        return len(var)!=0`
	`    else:`
	`        return True`
	`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:` C1
	`    assert False, f"Could not find '{filename}'"`
		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)

For Multi Targets	 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]` C2
	`    source = tar[1][1]`
	`    if source == 'Force Build Form':`
	`        ...`
		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	 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 variable (line 8 and 9). Next, we create an Assign node *assign* for each mapping pair from *Map* and copy the *assign* to the position of the *P* (line 11 and 12). Finally, we remove the *P* (line 13).

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

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

Table 2 shows the details of refactoring idiomatic code into non-idiomatic code of nine pythonic idioms.

B EVALUATION

To evaluate our approach, we study two research questions:

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

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

B.1 RQ1: Accuracy of Explaining Pythonic Idioms

Testing based verification To collect executed test cases of idiomatic code instances for 1,708,831 idiomatic code instances, we first use DLocator [3] statically analyze code to collect test cases that directly call the methods of the idiomatic code. Next, we execute the test cases before transforming the idiomatic code by installing required libraries of projects. As a result, there are 30,386 successfully executed test cases for 6,672 idiomatic code instances.

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

Since our approach involves two steps, detection and rewriting, we manually identify whether the test failure is due to wrongly detecting idiomatic code instances during the detection step or by incorrectly rewriting the idiomatic code instances during the rewriting step. The complete process is shown in Figure 1.

B.2 RQ2: Usefulness of Explaining Pythonic Idioms

B.2.1 Performance Comparison and Analysis. Table 3 shows the results of answer correctness and average completion time for each pythonic idiom and all pythonic idioms for G1 (control group) and G2 (experimental group). The last column lists the p-value of the Wilcoxon signed-rank test on the correctness difference and the completion time difference. For the answer correctness, G1 and G2 achieve 0.40~0.80 and 0.87~1 for nine pythonic idioms, respectively. The improvement of correctness is 25%~141.7% for different idioms. For all 27 questions, the overall correctness of G1 and G2 is 0.58 and 0.94, respectively. The improvement is 63.5% overall. And the P-value of *Correctness* column is less than 0.05 that shows the answer correctness of G2 is statistically significantly better than that of G1. Our results suggest that providing non-idiomatic code can improve the correct understanding of corresponding idiomatic code.

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

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 (0.57 and 0.63) respectively. According to Zhang et al. [4], list-comprehension is much more frequently used than set/dict-comprehension. Our prior idiom knowledge survey in Figure 2 also suggests that the participants generally know better and use list-comprehension more frequently than set/dict-comprehension. With the explanatory non-idiomatic code, the correctness gap between the three comprehension idioms becomes very small (1, 0.97 and 0.93 for list/set/dict comprehension respectively). Furthermore, the explanatory non-idiomatic code can speed up the understanding of all three comprehension idioms. For list comprehension that developers generally understand well, the understanding can still be speed-up by 19.4%.

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

• **truth-value-test idiom:** The improvement of correctness is 38.1%, with 17.8% speed-up. The truth-value-test involves a variety

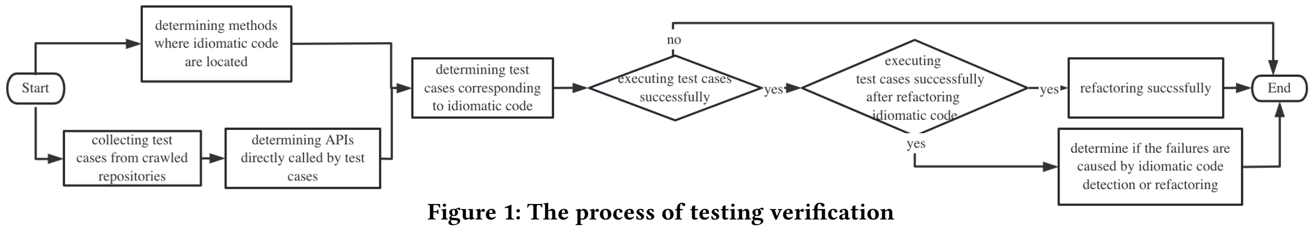

Start
determining methods where idiomatic code are located
collecting test cases from crawled repositories
determining APIs directly called by test cases
determining test cases corresponding to idiomatic code
executing test cases successfully
no
yes
executing test cases successfully after refactoring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

Figure 1: The process of testing verification

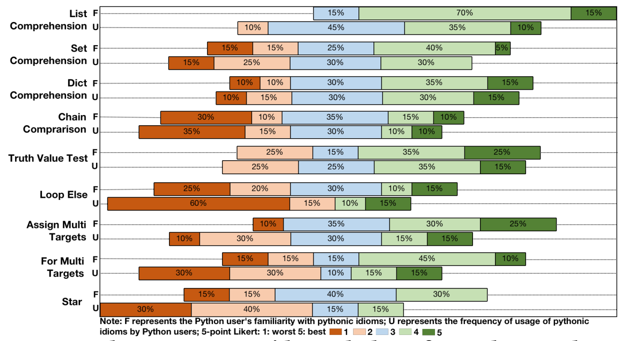


Figure 2: The participants' knowledge of 9 pythonic idioms

Table 3: Performance Comparison

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

of situations, e.g, any object like Call, BinOp and Attribute can be tested for truth value. Python users need deduce the value of the object and judge whether the current value defaults to false. It is challenging for G1 to understand the meaning of the idiom correctly and efficiently because 14 constants are defaulted to false (truth-value-test row of Table 2). Python users generally understand that None and 0 are considered false, but grasping all other situations is hard. For example, the value of the idiomatic code if d.expression is Decimal(0). 8 G1 participants answer wrongly but no one in G2 answers wrongly. The average completion time of G1 and G2 is 35.3s and 29s, respectively. The non-idiomatic code makes the G2 spend 17.8% less time than G1. As the non-idiomatic code contains two explicit statements (see the truth-value-test row of Table 2): "from decimal import Decimal" and "var in [None, False, '', 0, 0.0, 0j, Decimal(0), Fraction(0, 1), (), [], {}, dict(), set(), range(0)]", G2 participants can know the Decimal is a class with value 0.0 from decimal module and the value is default to false.

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

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

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

• **assign-multi-targets idiom:** The improvement of correctness is 107.1%, but with 6.1% slow-down in the understanding time. One common misconception G1 participants have for assign-multi-targets is the evaluation order of targets. For example, for the code "dummy2 = other = ListNode(0)", all G1 participants assume that "other" is assigned first, but "dummy2" is assigned first. The explanatory non-idiomatic code "tmp=ListNode(0); dummy2 = tmp; other = tmp" makes G2 participants avoid this misunderstanding. The other common misconception is the evaluation order of targets and value. For example, for the code "uc, ud = ud - q * uc, uc", 5 G1 participants think it is equal to "uc = ud - q * uc; ud = uc". It is because they do not realize the right-hand side (value of assign-multi-targets) is always evaluated before the left-hand side (targets of the assign-multi-targets). The explanatory non-idiomatic code “tmp = uc; uc = ud - q * uc; ud = tmp” helps G2 participants realize the correct evaluation order. These evaluation order misconceptions are also reflected in some Stack Overflow questions we investigated [1, 2].

• **star idiom:** The improvement of correctness is 47.4%, with 34.3% speed-up. Star can operate any iterable objects such as Subscript and Constant, and it can be used as parameters in the function call, the targets of for and targets and values of assignment statements. We find participants in G1 generally can answer question correctly for the * before a Subscript data object, but many G1 participants cannot understand other circumstances correctly. For example, for the idiomatic code “cv2.VideoWriter_fourcc(*‘mp4v’)”, we ask them about the number of arguments passed by the function call. 2 G1 participants think the code has syntax error and 3 G1 participants think the number of arguments is one, so the correctness is 50%. We interview them and find them do not know the star can be applied to the string constant or they think the number of elements to be unpacked is 1. The provided non-idiomatic code “cv2.VideoWriter_fourcc(‘m’,‘p’, ‘4’, ‘v’)” help Python users avoid such misunderstandings.

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] 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.

[4] 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).