RIdiom: Automatically Refactoring Non-idiomatic Python Code with Pythonic Idioms

APPENDIX A
APPROACH

Table I shows the detection rules and refactoring steps of anti-idiom code smells.

A. Detecting Anti-idiom Code Smells

1) List/Set/Dict Comprehension: The list-comprehension idiom is used for the list initialization (2nd row in Table I). The rule first finds an empty assignment statement $stmt_1$ (e.g., "dblist = []"). Then, it finds a for statement $stmt_n$ which iteratively adds elements to the target ("dblist") of $stmt_1$. There cannot be other statements using the target "dblist" of $stmt_1$ between $stmt_1$ and $stmt_n$ to lest the "dblist" is modified (i.e., $isNotUse(stmt_1.target, stmt_1, stmt_n)$). Since the $stmt_n$ corresponds to the comp node of the ListComp construct which only supports for clause and if clause, the rule checks whether $stmt_n$ satisfies the *MatchCompre* condition, i.e., if the $stmt_n$ corresponds to the syntax grammar of Comprehension. The detection rule for the non-idiomatic code of the set-comprehension and the dict-comprehension idiom are the same.

2) Chain Comparison: The chain-comparison "a op_1 b op_2 c ... y op_n z" is equivalent to "a op_1 b and b op_2 c and ... y op_n z" [**?**]. The non-idiomatic code of the chain comparison must be a $BoolOp$-and expression which contains at least two compare nodes. Moreover, the two compare nodes have the same operands. For example, for the expression "cp >= 178208 and cp <= 183983" (3rd row in Table I), the cp is the common operand of the two compare nodes, and the expression can be refactored as "183983 >= cp >= 178208".

3) Truth Value Test: The truth-value-test idiom is used for checking the "truthiness" of an object. Generally, when developers want to compare whether an object is equal or is not equal to a value, many programming languages use "==" or "!=" operator to achieve the functionality. In Python, any object can be directly tested for truth value, so developers do not need to use "==" or "!=" operator to test truth value. Python documentation specify the built-in objects in $EmptySet$ (e.g., [] and set()) are considered as false value. Therefore, if a statement directly compares an object to the element of $EmptySet$, it will be regarded as a non-idiomatic code of the truth value test. However, not all compare nodes are refactorable with truth value test. For example, "a!=[]" in "return a!=[]" cannot be refactored because "return a" changes the code semantic. According to Python syntax, the non-idiomatic code of truth value test corresponds to a test-type node. Therefore, our rule checks whether a compare node is the child of a test-type node, for example, the "runs([]) == []" is the child of an if-node "if runs([]) == []" (4th row in Table I). Since the if-node is a test-type node, the compare node "runs([]) == []" is refactorable to a truth-value-test.

4) Loop Else: The else clause of the loop statement is executed after the iterator is exhausted, unless the loop was ended prematurely due to a break statement. The non-idiomatic way of implementing a loop-else generally has an assignment statement $stmt_1$ to flag current state, a for statement $stmt_n$ which contains a statement s to change the current state and a break statement $stmt_j$ to end the loop, and an if statement $stmt_{n+1}$ after the for statement $stmt_n$ to check the current state to execute different operations. There are four circumstances: c_1 and c_2 complement each other, and c_3 and c_4 complement each other.

The c_1 satisfies the following semantic conditions: the semantic of the assignment statement $stmt_1$ is the same as the semantic of the test node of if statement $stmt_{n+1}.test$, and the semantic of assignment statement s is different from the semantic of $stmt_1$ where s and the break statement $stmt_j$ are at the same scope. These semantic conditions are designed because the non-idiomatic code of loop-else implies two execution paths (5th row in Table I): $stmt_1 \nrightarrow s$ and $stmt_1 \rightarrow stmt_{n+1}$ or $stmt_1 \rightarrow s$ and $stmt_j \nrightarrow stmt_{n+1}$.

The c_2 satisfies the following semantic conditions: the semantic of the assignment statement $stmt_1$ is the opposite of the semantic of the test node of the if-statement $stmt_{n+1}.test$, the if-statement $stmt_{n+1}$ has an else clause, and the semantic of the assignment statement s is the opposite of the semantic of $stmt_1$ where s and the break statement $stmt_j$ are at the same scope. The c_2 condition is a complement to the c_1 condition. If $stmt_{n+1}$ has an else clause and $stmt_{n+1}.test$ has the opposite semantic with $stmt_1$, it indicates that the else clause has the same semantic as $stmt_1$. Therefore, the code satisfying the c_2 condition is also refactorable to a loop-else. For example (5th row in Table 5), if we change $stmt_{n+1}.test$ "good_partition" into "not good_partition" and add an else clause to the if statement, the code satisfies the c_2 condition.

The c_3 satisfies the following semantic conditions: the semantic of the assignment statement $stmt_1$ is the same as the semantic of test node of the if-statement $stmt_{n+1}.test$, and the semantic of the if-statement s in the body of the loop statement $stmt_n$ is different from the semantic of $stmt_1$, and the body of the if-statement s contains the break statement $stmt_j$. The c_3 is a variant of c_1 and c_2. The c_1 and c_2 requires an assignment s to change the current state, but c_3 uses an if statement s to detect the change of the current state and break the loop, such as "if not good_partition: break".

The c_4 satisfies the following semantic conditions: the semantic of the the assignment $stmt_1$ is the opposite of the semantic of test node of the if-statement $stmt_{n+1}.test$, the

if-statement $stmt_{n+1}$ has an else clause, and the semantic of the test node of if statement $s.test$ in the body of the loop-statement $stmt_n$ is the opposite of the semantic of $stmt_1$ and the body of the if-statement s contains the break statement $stmt_j$. The c_4 complements c_3, in the same vein as c_2 complements c_1.

5) Assign Multiple Targets: The assign-multiple-targets idiom is to assign multiple values at the same time in one assignment statement. For several consecutive assignment statements, if an assignment statement $stmt_k$ does not use the result of an assignment statement $stmt_i$ before it, these assignment statements are refactorable to assign-multi-targets. When an assignment statement $stmt_k$ uses the result of the an assignment statement $stmt_i$ before $stmt_k$, the code usually is to swap variables by creating temporary variables. For such non-idiomatic code, it requires that the target of a statement $stmt_j$ between the $stmt_i$ and the $stmt_k$ is the same as the value of $stmt_i$. For example (third-to-last row in Table I), $stmt_k$ "d[e] = f" uses the target "f" of $stmt_i$ "f = d[0]", and the target "d[0]" of the $stmt_j$ "d[0] = d[e]" is the same as the value "d[0]" of the $stmt_i$ "f = d[0]". This sequence of assignments via a temporary variable can also be refactored with the assign-multiple-targets idiom.

6) Star in Function Calls: The star-in-function-call idiom is usually used to unpack an iterable to the positional arguments in a function call [**?**]. The non-idiomatic way of passing a sequence of arguments is that the subscript sequence of multiple consecutive parameters of a function call is an arithmetic sequence of the same variable. For example, "1, 2, 3" is an arithmetic sequence where the common difference is 1 for accessing the first, second and third element of "sys.argv" (second-to-last row in Table I). It can be refactored into "*sys.argv[1:4:1]".

7) For Multiple Targets: The non-idiomatic code of the for-multiple-targets idiom only contains one variable as the target of for statement p. The body of p uses the subscript expression to get elements of the variable. For example (the last row of Table I), the code uses "interval[0]" and "interval[1]" to get the elements of the variable "interval" inside the body of for loop. Instead, the elements of "interval" can be accessed using a for-multiple-targets idiom.

B. Refactoring with Pythonic Idioms

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

(1) **Copy(s, i)** copies the node s of non-idiomatic code to the position i of a node of idiomatic code. If the node at the position i is empty, we copy s into the position i. Otherwise, we insert s into the position i. Since a refactoring does not change the code semantics, many parts of non-idiomatic code can be copied to the resulting idiomatic code. For example, for the list-comprehension idiom (2nd row in Table I), both the target node item and the iter node cmplist of non-idiomatic code are copied to the corresponding target and iter position of the comprehension node respectively. For another example, for the chain-comparison idiom (3rd row in Table I) we copy operands of compare node of non-idiomatic code into the position of operands of a new compare node.

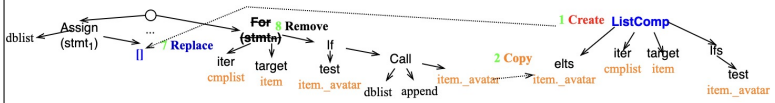
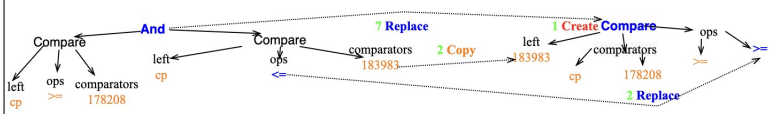
(2) **Create(s, *info)** builds the node of type s with information $*info$ where * represents any amount of information. To refactor non-idiomatic code into pythonic idioms, it is sometimes necessary to create some new AST nodes or elements which do not have the corresponding parts in the non-idiomatic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

(3) **Remove(s)** removes the node s from the AST of non-idiomatic code which is no longer needed in idiomatic code. Generally, refactoring non-idiomatic code into idiomatic code will reduce the lines or tokens of code. Therefore, it is natural to remove those no-longer-used nodes. For example, for the loop-else idiom (5th row in Table I), we need to remove the initial flag assignment "good_partition = True" and the flag-update statement "good_partition = False" which are no longer needed when the loop-else idiom is used. For another example, for the assign-multi-targets idiom (6th row in Table I), we remove assign statements from $stmt_2$ to $stmt_n$.

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

The 3rd column of Table I shows the refactoring steps to complete each pythonic idiom refactoring. The green line numbers shows the steps that are performed to refactor the examples of non-idiomatic code on the left into the idiomatic code on the right in the 2nd column of Table I. For example, to refactor the non-idiomatic code example into a list comprehension code (2nd row in Table I), we first create a ListComp node $comp$ and then traverse the for statement $stmt_n$ to copy its children to the $comp$ node (line 1-2), e.g., copy item._avatar to the position of $stmt_n.elt$ (i.e., elements to add to the list). Since $stmt_n$ and $stmt_1$ are at the same scope (line 6), we directly replace $stmt_1.value$ with $comp$ in and then remove $stmt_n$ (line 7-8). Finally, the new $stmt_1$ is the idiomatic code obtained through the refactoring. When $stmt_1$ and $stmt_n$ are at different scope (line 3), we do not perform the Remove operation for the $stmt_1$ because $stmt_n$ may not be executed after executing $stmt_1$, so we only replace $stmt_n$ with $stmt_1$ and then update the value of $stmt_n$ (line 4-5).

TABLE I: Examples of detection and refactoring of anti-idiom code smells

Idiom	Detection Rules and Examples of Code Pairs	Refactoring Steps
List/ Set/ Dict Compre-hension	$P = [stmt_1, stmt_n]$ *where* $stmt_1 = Assign \wedge stmt_1.value \in \{[], \text{set}(), \text{dict}(), \{\}\}$ $\wedge \text{isNotUse}(stmt_1.target, stmt_1, stmt_n) \wedge stmt_n = For \wedge \text{MatchCompre}(stmt_n)$ ($stmt_1$): dblist = [] ($stmt_n$): ~~for item in cmplist:~~ ~~if item._avatar:~~ ~~dblist.append(item._avatar)~~ 7 Replace ($stmt_1$): dblist = [item._avatar for item in cmplist if item._avatar] 2 Copy 	1: $comp$=Create("ListComp"/"SetComp"/"DictComp") 2: Traversing $stmt_n$, keep copying its children to $comp$ 3: **If** $stmt_1$ and $stmt_n$ have the different parents **then** 4: Replace($stmt_n$, $stmt_1$) 5: Replace($stmt_n.value$, $comp$) 6: **Else** 7: Replace($stmt_1.value$, $comp$) 8: ~~Remove~~($stmt_n$)
Chain Comparison	$P = p$ *where* $p = BoolOp \wedge p.op = And \wedge \exists(value_1, value_2)(value_1 = Compare$ $\wedge value_2 = Compare \wedge value_1 \in p.values \wedge value_2 \in p.values) \wedge \exists expr(expr \in \{$ $value_1.left, value_1.comparators[-1]\} \wedge expr \in \{value_2.left, value_2.comparators[-1]\})$ (p): cp >= 178208 and cp <= 183983 2 Copy 183983 >= cp >= 178208 7 Replace (com) 	1: com=Create("Compare") 2: Traversing $value_1$ and $value_2$ to copy and replace their children to com 3: **If** Num($P.values$)>2 **then** 4: Replace($value_1$, com) 5: ~~Remove~~($value_2$) 6: **Else** 7: Replace(P, com)
Truth Value Test	$P = p$ *where* $p.parent = test \wedge p = Compare \wedge \text{Num}(p.ops) = 1$ $\wedge \forall op(op \in p.ops \wedge op \in \{Eq,\ NotEq\} \wedge \{p.left,\ p.comparators[0]\} \cap EmptySet \neq \emptyset)$ if: runs([]) == [] 4 Copy if: not runs([]) (p) 7 Replace (node) 	1: Get $empty\_node$ which belongs to $EmptySet$ from {p.left, p.comparators[0]} 2: **If** $P.ops$ is Eq **then** 3: $node$=Create("Not") 4: Copy($empty\_node$, Index($node.operand$)) 5: **Else** 6: $node$= $empty\_node$ 7: Replace(P, $node$)

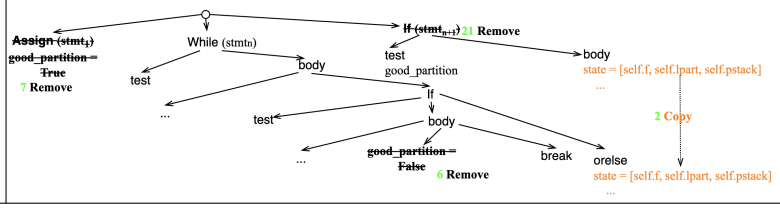
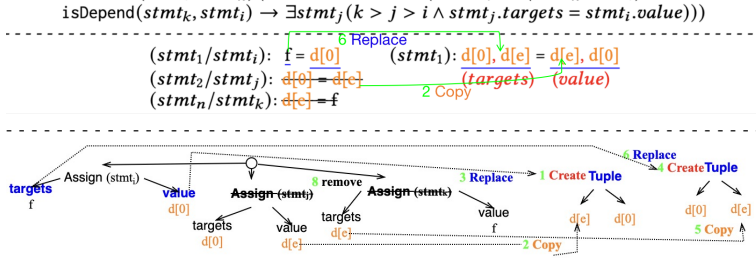
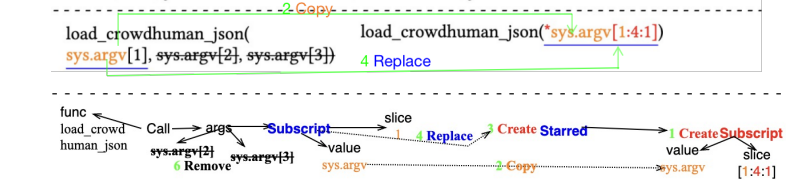
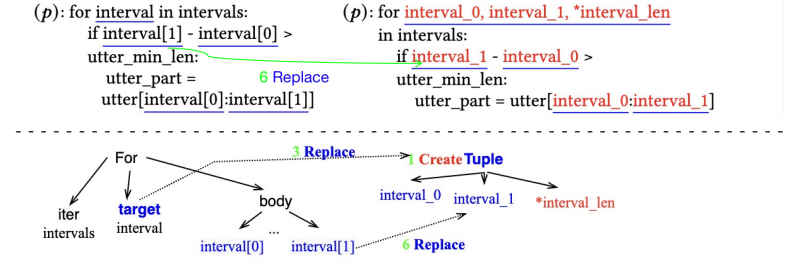
Loop Else	$P = [stmt_1, stmt_n, stmt_{n+1}]$ *where* $stmt_1 = Assign \wedge stmt_n \in \{For, While\} \wedge stmt_n.orelse = \emptyset$ $\wedge stmt_{n+1} = If \wedge (c_1 : \forall stmt_j(stmt_j = Break \wedge stmt_j \in stmt_n.body \rightarrow \exists s(s = Assign$ $\wedge stmt_j.parent = s.parent \wedge \text{SameSem}(stmt_1, stmt_{n+1}.test) \wedge \text{DiffSem}(s, stmt_1)))$ $\vee c_2 : \forall stmt_j(stmt_j = Break \wedge stmt_j \in stmt_n.body \rightarrow \exists s(s = Assign \wedge stmt_j.parent = s.parent$ $\wedge \text{OppositeSem}(stmt_1, stmt_{n+1}.test) \wedge stmt_{n+1}.else \neq \emptyset \wedge \text{OppositeSem}(s, stmt_1)))$ $\vee c_3 : \forall stmt_j(stmt_j = Break \wedge stmt_j \in stmt_n.body \rightarrow \exists s(s = If \wedge stmt_j \in s.body \wedge$ $s \in stmt_n.body \wedge \text{SameSem}(stmt_1, stmt_{n+1}.test) \wedge \text{DiffSem}(s.test, stmt_1)) \vee c_4 : \forall stmt_j($ $stmt_j = Break \wedge stmt_j \in stmt_n.body \rightarrow \exists s(s = If \wedge stmt_j \in s.body \wedge s \in stmt_n.body \wedge$ $\text{OppositeSem}(stmt_1, stmt_{n+1}.test) \wedge stmt_{n+1}.else \neq \emptyset \wedge \text{OppositeSem}(s.test, stmt_1))))$
	($stmt_1$): ~~good_partition = True~~
	($stmt_n$): while self.spread_part_multiplicity():
	self.db_trace("spread 1")
	if self.lpart >= ub:
	self.discarded += 1
	(s): ~~good_partition = False~~
	self.db_trace(" Discarding")
	self.lpart = ub - 2
	($stmt_j$): break
	($stmt_{n+1}$): ~~if good_partition:~~
	~~state = [self.f, self.lpart, self.pstack]~~
	~~yield state~~
	($stmt_n$): while self.spread_part_multiplicity():
	self.db_trace("spread 1")
	if self.lpart >= ub:
	self.discarded += 1
	self.db_trace(" Discarding")
	self.lpart = ub - 2
	break
	else:
	state = [self.f, self.lpart, self.pstack]
	yield state
	2 Copy
	 
		1: **If** c_1 **then**
		2: Copy($stmt_{n+1}.body$, Index($stmt_n.orelse$))
		3: **If** $stmt_{n+1}.orelse$ is not $None$ **then**
		4: Copy($stmt_{n+1}.orelse$, Index($stmt_j$)))
		5: **If** $stmt_1.targets$ does not occur in other statements in detection rules **then**
		6: ~~Remove~~(s)
		7: ~~Remove~~($stmt_1$)
		8: **Else If** c_2 **then**
		9: Copy($stmt_{n+1}.orelse$, Index($stmt_n.orelse$))
		10: Copy($stmt_{n+1}.body$, Index($stmt_j$))
		11: **If** $stmt_1.targets$ does not occur in other statements in detection rules **then**
		12: ~~Remove~~(s)
		13: ~~Remove~~($stmt_1$)
		14: **Else If** c_3 then
		15: Copy($stmt_{n+1}.body$, Index($stmt_n.orelse$))
		16: **If** $stmt_{n+1}.orelse$ is not $None$ **then**
		17: Copy($stmt_{n+1}.orelse$, Index($stmt_j$))
		18: **Else**
		19: Copy($stmt_{n+1}.orelse$, Index($stmt_n.orelse$))
		20: Copy($stmt_{n+1}.body$, Index($stmt_j$))
		21: ~~Remove~~($stmt_{n+1}$)

Table 5: Continued

Idiom	Detection Rules and Examples of Code Pairs	Refactoring Steps
Assign Multiple Targets	$P = [stmt_1, ..., stmt_n]$ *where* $\forall stmt_i (n+1 > i > 0 \wedge \texttt{Num}(stmt_i.targets) = 1)$ $\wedge \forall (stmt_i, stmt_k)(n+1 > k > i > 0 \wedge (\sim \texttt{isDepend}(stmt_k, stmt_i) \vee \texttt{isDepend}(stmt_k, stmt_i) \rightarrow \exists stmt_j (k > j > i \wedge stmt_j.targets = stmt_i.value)))$ 	1: *value*=Create("Tuple") 2: Traversing P to copy its children to *value* 3: Replace($stmt_1$.*value*, *value*) 4: *targets*=Create("Tuple") 5: Traversing P to copy its children to *targets* 6: Replace($stmt_1$.*targets*, *targets*) 7: **For** i from 2 to n **do** 8: Remove($stmt_i$)
Star in Func Call	$P = [arg_1, ..., arg_n]$ *where* $\forall arg_i (n+1 > i > 0 \wedge arg_i \in Call.args \wedge p_i = Subscript)$ $\wedge \forall (arg_i, arg_j)(n+1 > i,\ j > 0 \wedge arg_i.value = arg_j.value) \wedge \texttt{IsArithmeticSeq}(P)$ 	1: *subs*=Create("Subscript", P) 2: Traversing P to compute and create the slice node of *subs*, and copy the value of Subscript node value to the value of *subs* 3: *star*=Create("Starred", *subs*) 4: Replace(arg_1, *star*) 5: **For** i from 2 to n **do** 6: Remove(arg_i)
For Multiple Targets	$P = p$ *where* $p = For \wedge \texttt{Num}(p.target) = 1 \wedge \texttt{MatchSubscript}(p.target, p.body)$ 	1: Getting a variable mapping pair *Map*, where each element consists of a original variable and a new variable 2: *target*=Create("Tuple", Map) 3: Replace($stmt_1$.*target*, *target*) 4: **For** e in traverse($P.body$) **do** 5: **If** e in *Map* **then** 6: Replace(e, *Map*[e])

P represents non-idiomatic code; other variables representations come from Python documentation of abstract syntax grammar [16], e.g., *Assign* represents the Assign statement; $EmptySet = \{None, False, '', 0, 0.0, 0j, Decimal(0), Fraction(0,1), (), [], \{\}, set(), range(0)\}$.
$\texttt{MatchCompre}(stmt)$ returns true if $stmt$ corresponding to syntax grammar of Comprehension; $\texttt{isNotUse}(stmt_1.target, stmt_1, stmt_n)$ returns true if the targets of the $stmt_1$ is not used between $stmt_1$ and $stmt_n$; $\texttt{Num}(s)$ returns the number of elements in s; $c1, c2, c3, c4$ represent four different conditions; $\texttt{SameSem}(stmt_1,\ stmt_2)$ represents $stmt_1$ and $stmt_2$ have the same semantic; $\texttt{Opposite Sem}(stmt_1,\ stmt_2)$represents $stmt_1$ and $stmt_2$ have the opposite semantic; $\texttt{DiffSem}(stmt_1,\ stmt_2)$represents $stmt_1$ and $stmt_2$ have the different semantic; $\texttt{IsDepend}(stmt_1, stmt_2)$ represents $stmt_1$ is dependant on $stmt_2$ or $stmt_1$ depends on the result of $stmt_2$; $\texttt{IsArithmeticSeq}(P)$ returns true if the sequence consisting of slice of all elements of P is an arithmetic sequence; $\texttt{MatchSubscript}(s, t)$ represents t uses s with the Subscript (i.e., "s[]"), and the subscript of s is a constant. Orange, red, strikethrough and blue underlined text represent Copy, Create, Remove and Replace operations respectively; Green line numbers shows the steps that are performed to refactor the examples of non-idiomatic code on the left into the idiomatic code on the right in the 2nd column.