Automated Refactoring for Pythonic Idioms with Differential Testing

1 Introduction

Python is widely used across various domains due to its rich libraries and flexible syntax. However, many developers struggle with writing **idiomatic** Python code, which can compromise readability, make maintenance more challenging, and sometimes even degrade performance. To address these issues, we have developed an **automated refactoring** technology. This technology transforms Python code into more **Pythonic** code, enhancing both readability and performance. To verify that the refactoring does not alter the functionality of the code, we have implemented **differential testing**. This method confirms that the code behaves the same before and after refactoring, thus securing the stability of the refactoring process. This report will discuss the necessity and effectiveness of this approach and examine how automated refactoring can improve the quality of Python code.

2 Related Work

2.1 Making Python Code Idiomatic by Automatic Refactoring Non-idiomatic Python Code with Pythonic Idioms - Zhang et al. (2022)

This study presented the first tool to automatically transform non-idiomatic Python code into idiomatic Python code. They analyzed 7,638 Python projects on GitHub, identifying **9 types** of 'Pythonic Idioms' that frequently occurred and could be converted from non-idiomatic code. Based on this, they defined specific syntactic patterns and developed a method of refactoring code through **abstract syntax tree (AST)** rewriting operations. Moreover, they applied these refactorings to actual code to verify the accuracy and usefulness of their tool. This research provides an essential foundation for our study, inspiring the design and implementation of our refactoring tool.

2.2 title2 - author

...

3 Method?

3.1 Automated Refactoring

In this project, we laid the groundwork for the implementation of our refactoring tool by referencing the study of Zhang et al. and various literatures on Python idioms. Specifically, we selected **10 types** of refactorings that could enhance the performance and readability of Python code. These refactorings were implemented using the **Abstract Syntax Tree (AST)**, which provides a structural representation of source code. The AST is an ideal tool for systematically analyzing and modifying each element of the code. Through this, we developed a methodology that could automatically improve the code into a more Pythonic form.
Details of this part are all on **refactor.py**. For better understanding, we will write the (relevant code line) after the description of each part. For example, we inherited the NodeTransformer class and performed automated refactoring.**(413)**
When developing our automated refactoring tool, we assumed that the **source code is free of any errors**. This assumption allows us to focus solely on the refactoring process without the need to address or correct pre-existing code errors.

(a) Collection Comprehension

```
# Python codes
```

...

(b) Chaining Comparisons ...

(c) ?? ...

(d) Merge Append ...

(e) Test Empty Collection

```
# original
if A==[]:
    print(1)
elif B!=():
    print(2)
# refactored
if not A:
    print(1)
elif B:
    print(2)
```

Test collection changes a comparison with Eq(==) or NEq(!=) which one side is empty collections (list, tuple, dictionary) or empty string to a form without a comparison. Therefore, we overwrite **visit_Compare**, and change the content of the node to #refactored in code snippet when the above conditions are satisfied.(7)

(f) Merge If

```
1  # original
2  if A:
3      if B:
4          if C:
5              print(2)
6          else:
7              print(3)
8  # refactored
9  if A and B:
10     if C:
11         print(2)
12     else:
13         print(3)
```

Merge If combines multiple If statements into one using the and operator when they appear consecutively. Therefore, we overwrite **visit_If** and change the content of the node to #refactored in code snippet when the above conditions are satisfied.
However, it is crucial to consider cases where the if statement **appears with else** together, which would make merging inappropriate. For example, consider the lines 2, 3 and 5 of the code snippet. To handle this case, when a If node has **node.orelse**, we don't apply the merge.(30)
Also, to merge 3 or more mergeable If statements, we first implement **generic_visit**.(459)

(g) If Expression

```
1  # original
2  if A:
3      x=1
4  else:
5      x=2
6  if A:
7      print(1)
8  else:
9      print(2)
10 # refactored
11 x = 1 if A else 2
12 print(1 if A else 2)
```

If Expression changes a general If-else syntax to IfExpression when the contents of the If-else are **different only in value and perform the same function (assignment, print)**. Therefore, we overwrite **visit_If** and change the content of the node when the above conditions are satisfied.(47)

(h) to Enumerate

```
1  # original
2  for i in range(len(players)):
3      print(i, players[i])
4      print(players[i-1])
5      players[i] = 1
6  # refactored
7  for i, item in enumerate(players):
8      print(i, item)
9      print(players[i - 1])
```

```
10      players[i] = 1
```

to Enumerate always changes the form of the range (len(list or tuple)) in the for statement to enumerate (list or tuple). Therefore, we overwrite **visit_For**.(67)
For convenience, we unify the variable to store the element of the list as **item**. In addition, by overwriting **visit_Subscript**, we use **item** instead of the list **element by using current index** inside the for statement in the source code.(87) To implement this function, we defined a dictionary called **self.toItem**.(415) At the beginning of the visit of the for statement, add **(list name):(index)** to the self.toItem(77), and pop when all the visit of the for statement is over.(430)
However, it is crucial to consider cases where accessing an element of an index **different from the current index** of the list or assigning a new value **to the current index element** of the list. To handle these cases, we check that each node satisfies the above state in visit_Assign and visit_Subscript, and if so, we did not proceed with the change to item.(89, 439)

(i) Return Boolean Statement

```
1  # original
2  def func(A):
3      if A:
4          return True
5      else:
6          return False
7  # refactored
8  def func(A):
9      return A
```

Return Boolean Statement is possible only when returning a Boolean value with different internal branches with if-else statement. Therefore, we overwrite **visit_If**, and change the content of the node when the above conditions are satisfied.(104)

(j) Multiple Assign

```
1  # original
2  a=2
3  b=3
4  c=4
5  d=c+1
6  # refactored
7  a,b,c=2,3,4
8  d=c+1
```

Multiple assistance can occur in any part of the Python code. The strategy involves modifying the **generic_visit** method to find out all nodes that contain a 'body' attribute, and **traverse node.body** which includes the potential for multiple assignments.(417)
However, it is crucial to consider cases where variables from previous assignments are used in subsequent ones, which would make merging inappropriate. For example, consider the **c** and **d** on the lines 4 and 5 of a code snippet. To handle this case, we apply always storing the previous element and comparing it with the current element while traverse all node.body.(147)

3.2 Differential Testing

...

```
# Python codes
```

...

4 Evaluation

4.1 Efficiency

...

4.2 Readability?

...

4.3 Differential Testing

...

5 Threat to Validity

5.1 Handling Corner Cases in Automated Refactoring

One of the threats to validity in the development of our automated refactoring tool is the **potential failure to perfectly handle specific corner cases**. Although we attempted to cover all conceivable counterexamples theoretically, it is practically challenging to anticipate every scenario due to the limits of imagination. To overcome this limitation, we adopted an approach of applying automated refactoring to **existing Python codes**, examining the outcomes to identify and modify counterexamples. In particular, we utilized various code examples from programming problem platforms like **Baekjoon** to assess the applicability and usefulness of our tool. This process has allowed us to enhance the robustness of our refactoring tool in handling diverse real-world use cases.

5.2 Hmm

...

6 Discussion & Conclusion

...