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 A Peer Architecture for Lightweight Symbolic Execution - Alessandro Bruni, Tim Disney, Cormac Flanagan (2011)

This study presents a lightweight, library-based approach to symbolic execution, leveraging the dynamic dispatch capabilities of object-oriented languages like Python. Instead of creating a new interpreter or compiler, their approach embeds a symbolic execution engine within the target program's process. This engine uses proxy objects to simulate

symbolic values and track program execution paths. This provided a direction on how to perform symbolic execution for testing the functionality of functions.

3 Approach

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

```
# original
A=[]
for i in range(10):
    A.append(i)
for i in range(10):
    A.append(i*2)
# refactored
A = []
A += [i for i in range(10)]
A += [i * 2 for i in range(10)]
```

Collection comprehension transforms the declaration of {lists, dictionaries and sets} into a more Pythonic and efficient style. The node transformer class **CodeReplacer** calls the function **perform_comprehension** for each function definition and module level to perform all possible comprehensions. In the refactored code, the list A is created and extended using list comprehensions, allowing multiple comprehensions to be combined effectively.

(b) Chaining Comparisons

```
# original
if d>e or (b>=c and d<=c and b<a and e>c) or e>10 and c>e:
    print("OK")
# refactored
if d > e or (e > c and a > b >= c >= d) or (c > e > 10):
```

In the refactored code, the original complex condition is broken down into more manageable parts by using Python's ability to chain comparisons directly. When **Visit_If** is called, it invokes the function **transform_chaining_comparisons**, which checks all possible combinations and refactors them into the simplest equations possible.

(c) Multiple Equality Comparisons

```
# original
name = "Joe"
if name == "Casey" or name == "Joe" or name == "Mike":
    print("yes")
# Refactored
name = 'Joe'
if name in ['Casey', 'Joe', 'Mike']:
    print('yes')
```

Multiple Equality Comparisons code auto-refactoring simplifies the process of comparing a variable against multiple values. The refactored version replaces these with a membership test using the 'in' keyword, checking if the variable is within a list of values. The node transformer class **CodeReplacer** applies the function **transform_equality_comparisons** for every **Visit_If**.

(d) Merge Append

```
# original
A = []
A.append(1)
A.append(2)
A.pop()
A.append(3)
A.append(4)
# Refactored
A = []
A += [1, 2]
A.pop()
A += [3, 4]
```

Merge multiple append optimizes the code by combining consecutive append calls into a single operation using list addition. The function **transform_list_appends**, which is responsible for it, checks every append call within a module and function, and merges them into a single **augAssign** when more than two appends are detected, utilizing AST. The original sequence of **A.append** calls is replaced by **A += [1, 2]** and **A += [3, 4]**, consolidating the append operations while maintaining the same functionality.

(e) Test Empty Collection

```
# original
if A==[]:
    print(1)
elif B!=():
    print(2)
# refactored
```

```
7  if not A:
8      print(1)
9  elif B:
10     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. For example, the lines 9 and 10 in code snippet. 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

The purpose of refactoring is to improve the structure of the code, enhancing readability and performance, without changing the functional behavior of the code. If the refactored code behaves differently from the original code, it cannot be considered correctly refactored. Therefore, it is crucial to ensure that the refactored Python source code operates identically to the original code. In this section, we perform differential testing to verify this, using two methods to confirm that the refactored code operates the same as the original one.

(a) Function Test

The first method is to test whether the functions implemented in the code produce the same return values. Since the behavior of a function depends on the input parameters, a variety of values should be used as input parameters. We used symbolic execution to obtain the input parameters that directly execute the modified parts. The purpose of the function test is to verify whehter the return values of the functions differ by generating all possible combinations of input parameters for every conditional statement and branch within the code through symbolic execution, thus covering as much of the code as possible. If the return values differ for a specific input, it indicates that the refactoring was not performed correctly.

Symbolic execution involves executing the target program not with concrete test inputs but with symbolic proxy objects set as input parameters. These proxy objects allow the symbolic engine to track and analyze the execution flow of the program. In this process, it is possible to observe how the program manipulates the input parameters and how these manipulations affect the rest of the program. Unlike many other programming languages, Python does not provide type hints normally, making it difficult to determine the types of input parameters based solely on the information provided by the function. In order to solve this problem, we created an 'AnyProxy' class and used it as the input parameters for the functions. This class helps to generalize input types initially, and then, based on the behavior of the program, it differentiates these inputs into specific types for

further processing.

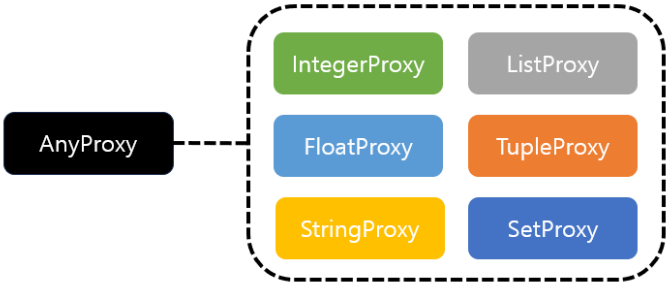

Figure 1. The proxy classes differentiated from AnyProxy class

Figure 1 illustrates how the AnyProxy class we implemented branches into specific classes. AnyProxy is initially set as the input parameters of a function and, during the execution of program, differentiates into specific types of proxy objects through various operations and method calls. The AnyProxy object differentiates into a specific type of proxy object in the following situations.

1. Operation with a specific type: When the AnyProxy object performs an operation with data of a specific type, it transforms into a proxy object corresponding to that type. For example, if an AnyProxy object performs an addition operation with an int type, it differentiates into an IntegerProxy object.

```
def get_proxy(self, other):
        if isinstance(other, (int, IntegerProxy)):
            return IntegerProxy(Int(self.name)), other
        elif isinstance(other, (float, FloatProxy)):
            return FloatProxy(Real(self.name)), other
        elif isinstance(other, (str, StringProxy)):
            return StringProxy(String(self.name)), other
        ...
```

2. Specific method call: When the AnyProxy object encounters a method that is unique to a specific type, it transforms into a proxy object for that type. For example, it the AnyProxy object calls 'append' method, which is unique to the list type, it differentiates into a ListProxy object.

```
def __setitem__(self, index, value):
        return ListProxy(IntVector(self.name, self.length), self.name)
    .__setitem__(index, value)
```

(b) Execution Test

The second method involves verifying that the refactored Python code behaves identically to the original Python code by directly executing both codes and comparing their

standard stream outputs. This is achieved using the Python 'subprocess' module to compare the values output to 'stderr' and 'stdout'. If the results differ, it indicates that there were some problems during the refactoring process, and an error is returned.

```
def test_refactored_code(original_path, test_path):
    cur_dir = os.getcwd()
    test_case_path = os.path.dirname(original_path)
    ori_file_name = os.path.basename(original_path)
    test_file_name = os.path.basename(test_path)
    os.chdir(test_case_path)
    cmd1 = ['python', ori_file_name]
    result1 = subprocess.run(cmd1, capture_output=True, text=True)
    cmd2 = ['python', test_file_name]
    result2 = subprocess.run(cmd2, capture_output=True, text=True)

    os.chdir(cur_dir)
    try:
        assert result1.stdout == result2.stdout
        assert result1.stderr == result2.stderr
    except Exception as e:
        print("Error while testing stdout and stderr", e)
```

4 Evaluation

We evaluated our tool from two main perspectives. The first perspective is to determine whether the refactored code is more efficient to execute compared to the pre-refactored code. To assess this, we compared the execution times of various operations to observe any improvements in efficiency. The second perspective is to verify whether the refactored code consistently produces the same results as the original code. This aspect is crucial for ensuring the accuracy and reliability of the code. Thus, we examined whether the refactored code delivers performance enhancements while fully meeting the original functional requirements. Through this evaluation, we aimed to confirm the overall performance and reliability of our tool.

4.1 Efficiency

Figure 2 compares the execution times of various operations before and after refactoring, indicating the efficiency gains achieved. If each is called before code and after code, both were executed 500 times and t-tested on the execution result. Overall, the refactoring has led to significant improvements in execution times across most operations. For example, the merge append operation exhibits a notable reduction in execution time, almost halving from the initial value. Similarly, other operations such as multiple assignments, testing empty collections, and chaining comparisons demonstrate substantial performance enhancements. These improvements illustrate the effectiveness of the refactoring process in optimizing code execution and enhancing overall efficiency.

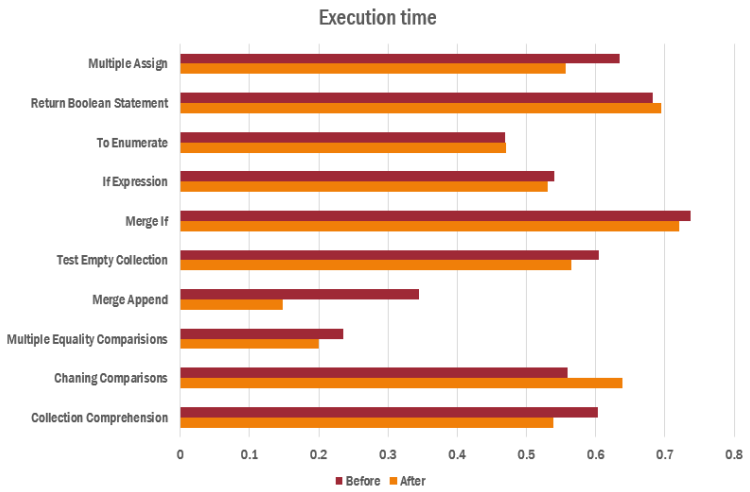

Figure 2. Comparing Execution Time Before and After Refactoring

4.2 Differential Testing

Figure 3 shows the result of our differential testing in total 44 target pytest functions. The upper 8 files with integer are from baekjoon problem. Next 12 files starts with 'diff' are from our sample test codes we created based on the 10 refactoring targets. Last 5 files are from our CS453 assignment 4: mutation. The result shows that all 44 target functions produce the same results for the same input before and after refactoring. This means that all 44 functions passed differential testing without exception.

```
============================= test session starts =============================
platform win32 -- Python 3.10.9, pytest-8.2.0, pluggy-1.5.0
rootdir: C:\Users\ckalsgh56\Desktop\Auto_refactoring-differential_testing
plugins: anyio-3.5.0
collected 44 items

test_cases\test_1011.py .                                                [  2%]
test_cases\test_11054.py .                                               [  4%]
test_cases\test_1111.py .                                                [  6%]
test_cases\test_12015.py .                                               [  9%]
test_cases\test_2110.py .                                                [ 11%]
test_cases\test_2162.py .....                                            [ 22%]
test_cases\test_9252.py .                                                [ 25%]
test_cases\test_diff_example0.py .                                       [ 27%]
test_cases\test_diff_example1.py .                                       [ 29%]
test_cases\test_diff_example10.py .                                      [ 31%]
test_cases\test_diff_example11.py .                                      [ 34%]
test_cases\test_diff_example2.py .                                       [ 36%]
test_cases\test_diff_example3.py .                                       [ 38%]
test_cases\test_diff_example4.py .                                       [ 40%]
test_cases\test_diff_example5.py .                                       [ 43%]
test_cases\test_diff_example6.py .                                       [ 45%]
test_cases\test_diff_example7.py .                                       [ 47%]
test_cases\test_diff_example8.py .                                       [ 50%]
test_cases\test_diff_example9.py .                                       [ 52%]
test_cases\test_law.py ..                                                [ 56%]
test_cases\test_lazy.py .....                                            [ 68%]
test_cases\test_math.py ...                                              [ 75%]
test_cases\test_simple1.py .......                                       [ 90%]
test_cases\test_simple2.py ....                                          [100%]

============================= 44 passed in 0.40s ==============================
```

Figure 3. Result of 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 Covering Other Type of Inputs

In the function testing part of the differential testing, we did not cover any type existed in Python. This is because there are limitations in the Z3 solver and challenges in type inference system of Python. The Z3 solver, while powerful, has limitations in handling complex user-defined types. Additionally, Python's dynamic typing and lack of type hints complicate accurate type inference. Therefore we were unable to implement code that covers all types. We can only create test cases for certain function which get specialized input parameters. These constraints may affect the generalizing. However, we expect that, with more cases' testing, the range of covering can be improved.

5.3 File System Differential Testing

When performing differential testing to determine if program execution is identical, it is also necessary to verify that the file system used by the program is the same. This involves ensuring that the state of the files and directories accessed or modified by the program remains consistent between the original and refactored versions. If the file system differs, differences in program behavior may generate due to environmental discrepancies. However, our differential testing only verifies the consistency of function return values and standard outputs, lacking a mechanism to ensure the consistency of the file system state. We plan to implement this in the future to eliminate differences in program behavior caused by environmental inconsistencies.

6 Feedback

6.1 Benchmark Codes

...

6.2 Random Test Generation

Here are examples of code to test the provided functions and code using random testing. We used 100 random inputs for testing. Similar to the symbolic execution approach, for execution tests, we compared the standard output and standard error. For function tests, we wrote code to compare the return values of the functions using pytest. While

symbolic execution required significant computational effort and time to generate a small number of test cases, random testing was able to generate a much larger number of test cases in a shorter period. However, there is a possibility that random testing might not cover all the code, impacting its coverage effectiveness.

Simple example

```python
import importlib.util
import pytest
from hypothesis import given, strategies as st

ori_module_spec = importlib.util.spec_from_file_location('ori_module', './examples/simple/diff_example2.py')
test_module_spec = importlib.util.spec_from_file_location('test_module', './updated/simple/diff_example2.py')
ori_module = importlib.util.module_from_spec(ori_module_spec)
test_module = importlib.util.module_from_spec(test_module_spec)
ori_module_spec.loader.exec_module(ori_module)
test_module_spec.loader.exec_module(test_module)

@given(st.lists(st.tuples(st.integers(), st.integers(), st.integers(), st.integers()), min_size=100, max_size=100, unique=True))
def test_if_expression_random(inputs):
    for a, b, c, d in inputs:
        assert ori_module.if_expression(a, b, c, d) == test_module.if_expression(a, b, c, d)
```

Baekjoon example

```python
import subprocess
import sys
import warnings
from hypothesis import given, settings, strategies as st
from pytest import PyAssertRewriteWarning

warnings.filterwarnings("ignore", category=PytestAssertRewriteWarning)

def execute_script(script_path, *args):
    result = subprocess.run(
        [sys.executable, script_path] + [str(arg) for arg in args],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )
    return result.stdout.decode(), result.stderr.decode()

def compare_outputs(original_output, updated_output):
    return original_output == updated_output

@settings(max_examples=100, deadline=None)
@given(st.integers(min_value=1, max_value=100), st.integers(min_value=1, max_value=100))
def test_main(x, y):
    if x > y:
        x, y = y, x

    original_file = "examples/backjoon/1011.py"
```

```
    updated_file = "updated/backjoon/1011.py"

    original_stdout, original_stderr = execute_script(original_file, x, y)
    updated_stdout, updated_stderr = execute_script(updated_file, x, y)

    assert compare_outputs(original_stdout, updated_stdout), f"stdout differs for inputs: x={x}, y={y}"
    assert compare_outputs(original_stderr, updated_stderr), f"stderr differs for inputs: x={x}, y={y}"

if __name__ == "__main__":
    import pytest
    pytest.main()
```

Symbolic execution and random testing are two prominent testing methodologies, each with its own advantages and disadvantages. Symbolic execution offers high coverage and the ability to generate specific input values, making it beneficial for uncovering boundary conditions and logical errors. However, it faces scalability issues and challenges in handling complex environments, making it difficult to apply to large or intricate programs. On the other hand, random testing is simple to implement and highly scalable, but it may fall short in terms of coverage and efficiency, potentially missing critical paths or generating less meaningful test cases.

When choosing a testing methodology, it's essential to consider the characteristics of the program and the goals of the testing process. For instance, symbolic execution may be more advantageous for identifying critical boundary conditions and logical errors, whereas random testing could be more suitable for quickly evaluating a wide range of input values. By appropriately combining both methodologies, one can establish a more effective and comprehensive testing strategy.

7 Discussion

...

8 References

...