Refactoring Tutor: An IDE Integrated Tool for Practicing Key Techniques to Refactor Code

by

Mario Leyva

S.B. Computer Science and Engineering
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

© 2023 Mario Leyva. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by:	Mario Leyva Department of Electrical Engineering and Computer Science May 12, 2023
Certified by:	Robert C. Miller Distinguished Professor of Computer Science Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Refactoring Tutor: An IDE Integrated Tool for Practicing Key Techniques to Refactor Code

by

Mario Leyva

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Refactoring code is an important skill to become a competent software engineer, however it is usually never explicitly taught in coding intensive courses. Even though engineers in academia and industry agree refactoring is important, most novice programmers are unaware of the code smells they should avoid when writing code. This thesis discusses a novel tutoring system to assist novice programmers with refactoring. This tool provides refactoring exercises to students in an introductory programming class. The tutor exposes students to various types of code smells and has them deliberately practice how to refactor. The tutor infrastructure has proven to be robust to several refactoring exercises. Based on a user study involving the students and the staff members from 6.1010: Fundamentals of Programming, the tutor infrastructure has shown to be robust to bugs and staff feedback. The tutor shows promise, but further studies with more students are necessary to evaluate its effectiveness on teaching student refactoring.

Thesis Supervisor: Robert C. Miller
Title: Distinguished Professor of Computer Science

Acknowledgments

Completing this thesis was not possible if it were not for the invaluable mentorship of my thesis advisor, Robert C. Miller. I am indebted to Prof. Rob Miller for all the guidance, support, and patience that helped me complete this project. I appreciated all the time and effort that was made to make sure my MEng went smoothly. I am rather lucky to have had such a great advisor.

I would also like to especially thank Gaby Ecanow. It was tremendously helpful having someone who was in the same boat as me throughout the MEng. I am grateful for all help getting started and especially appreciate the responsive texts on questions I had about the tutor.

The people who were courageous enough to try the refactoring tutor were a blessing; this included the TAs, LAs, and students of 6.1010. I thank them for providing me with crucial feedback and ideas to improve the tutor.

Last but not least, I would like to thank my family and friends, especially my parents. I cannot, in any way, repay all the time and effort my parents invested in me. Without them, I wouldn't have gotten through MIT, even less so accepted. The thought of returning the favor and making them proud is what has gotten me through so many sleepless nights and stressful times during my time at MIT.

Contents

1 Introduction 13

2 Related Work 17

2.1 Code Smells 17

2.2 Why Refactoring? 18

2.3 Deliberate Practice 18

2.4 Teaching Refactoring 19

3 Code Smells 21

3.1 Boolean Laundering 21

3.2 Line too long 22

3.3 Variable Renaming 25

3.4 Variable Naming Inconsistencies 26

3.5 Magic Numbers 27

3.6 Code Hoisting 29

3.7 Conditional Complexity 30

3.8 Excessive Nesting 32

3.9 Lack of Idiomatic Syntax 35

4 Concept Map 37

4.1 Level 0 Concept Group 37

4.2 Level 1 Concept Groups 38

4.3 Level 2 Concept Groups 39

5 Design 41
5.1 Exercise Authors . . . 41
5.2 The Student . . . 42
6 Implementation 49
6.1 Exercise Checking . . . 51
6.2 Diff Checking . . . 52
6.3 Diff Matching . . . 54
6.4 Exercise Metadata . . . 57
6.4.1 diffConfiguration . . . 60
6.4.2 Yanked Insertions . . . 61
6.4.3 Transformers . . . 63
6.4.4 Triggered Hints for Diffs . . . 66
6.5 Test Cases . . . 67
7 Deployment 69
7.1 First Deployment . . . 69
7.2 Second Deployment . . . 70
8 Evaluation 73
8.1 Student Results . . . 73
8.1.1 Boolean-laundering-if-statements . . . 75
8.1.2 Single-character-variable-name . . . 80
8.1.3 Single-character-variables-compound-interest . . . 83
8.1.4 Register-user-never-nest . . . 85
8.1.5 Tuple-unpacking-in-loops . . . 89
8.2 Staff Results . . . 91
8.3 Staff Feedback . . . 92
8.3.1 Hints . . . 93
8.3.2 Exercise Description . . . 94
8.3.3 Exercise Bugs . . . 95

8.4 Infrastructure Bugs 96

9 Discussion 99

9.1 Future Improvements 100

10 Conclusion 103

List of Figures

4-1 Refactoring Tutor Concept Map 38

5-1 Exercise Files 42
5-2 Refactoring Tutor Concept Map Progress 43
5-3 Code Hoisting Exercise 44
5-4 Running Test Cases on Refactoring Tutor 46
5-5 Submitting Exercise before running Test Cases 46
5-6 Failing Exercise on Refactoring Tutor 47
5-7 Completing Exercise on Refactoring Tutor 47

6-1 Refactoring Tutor Architecture 50
6-2 Diff Checker 54
6-3 Matching Student And Staff Diffs 56
6-4 Yanking Insertions in Magic Number Exercise 62

8-1 Exercises Completed by students in 6.1010 74
8-2 Passed and Failed Attempts per Exercise from students 75
8-3 Number of Exercises completed by staff 92
8-4 Passed and Failed Attempts per Exercise from staff 93

Chapter 1

Introduction

Code refactoring has always been an important aspect of software engineering. It is an essential skill when writing any code, whether that is in industry, school, or open source projects. Ensuring code is easy to read, maintainable, and ready for changes improves the software development process for both people interacting with a given code-base and the future author. Refactoring also results in less time spent on deciphering messy code. Undoubtedly, there are benefits to refactoring no matter where code is produced. Understanding when refactoring code is necessary even helps prevent writing incoherent code in the first place.

However, refactoring code is typically not explicitly taught in introductory CS classes. Unlike performance or testing, refactoring isn't a focus in CS classes. Usually, programmers gain skills for refactoring code as they work on exercises and projects in school or in industry. Getting a head start on learning to refactor would help minimize poor code feeding into code-bases, some of which affect important computer systems. Therefore, introducing the concepts of refactoring and teaching students about very common ways to refactor code along with how to recognize bad code smells is paramount during introductory CS classes. Throughout this thesis, different types of code smells will be mentioned. Definitions and examples of such code smells can be found in chapter 3 for reference.

The approach for this project revolves around directly providing students from an introductory programming course with a refactoring tutoring system. Most introduc-

tory classes have hundreds of students, therefore instructors typically can't address refactoring issues for every student. With an automated tutoring system however, every student would be able to get familiar with refactoring concepts and practice with the provided exercises. The main target for the refactoring tutor is 6.101: Fundamentals of Programming, one of the first programming classes in the computer science curriculum at MIT. In this class, refactoring is an essential skill since the programs students write become complex quickly. Since the class is purely taught in the Python programming language, the tutor focuses on refactoring in Python. However, most of the concepts that are focused on extend beyond Python and can be applied to other programming languages.

An essential design principle that we had in mind as we created the refactoring tutor was language-agnosticism. As mentioned in the previous paragraph, the tutor currently supports Python refactoring, but the infrastructure should be capable of supporting refactoring exercises in other programming languages too. Perhaps in the future, the tutor will be extended to support TypeScript so that it could be used in 6.102: Software Design (formerly 6.031).

The refactoring tutor was designed to teach low level skills about refactoring, which relies on recognizing code smells and non-idiomatic code style, through repetitive practice via exercises. The goal of the tutor is to provide students with plenty of practice so that the moment they write or read code, they will be able to quickly diagnose code smells and know which refactoring techniques to apply. In order to effectively teach students how to refactor, examples with code smells were curated with advice from CS instructors and the current literature regarding best code practices. Such exercises involved isolated pieces of code that focused in on a specific concept. Many different exercises were created that ranged from the simplest refactoring skills such as removing boolean laundering to the more complex skills such removing excessive nesting and code hoisting.

Refactoring spans many different skills. The skills we are tackling are focused on refactoring code in a general sense, however other researchers instead focus on system design and refactoring at a higher level. They focus more on object oriented

programming (OOP) and evaluation of such systems and exactly how one should go about refactoring. Specifically, they bring forth the concept of Unified Modelling Language (UML), which is a standardized modelling language that helps engineers develop software systems. UML consists of diagram and figures that engineers can use in order to develop, visualize, and document software systems. Using UML, engineers can then better understand how systems should be refactored if necessary. This is referred to as model refactoring and is the focus of many research projects. There even are some projects that have been successful at automatically creating UML diagrams given a code-base (Mens, 2007).

UML and model refactoring are out of scope for this project, as the audience for such refactoring is mostly meant for more advanced software engineers. This thesis focused on source code refactoring, which is more low level and concerned with refactoring smaller pieces of code compared to entire code-bases. Additionally, novice programmers should understand how source code refactoring should be done before they approach refactoring larger code systems.

There are two main users involved with the tutor: the students and exercise authors. As already mentioned, students complete exercises and learn about refactoring, using a popular integrated development environment (IDE), VSCode. Exercise authors are responsible for creating exercises for the students, which involves creating the initial code for students to refactor as well as the their solution (known as the staff solution). In order for exercises to be robust, the exercise authors can fine tune the exercises using diff configuration, transformers, and triggered hints to allow for unexpected solutions from the student which are still refactored correctly.

The refactoring tutor was built on the Praxis Tutor infrastructure, a programming tutorial system. The existing infrastructure has support for exercises to learn Typescript or Java, however I added support for Python. Also, a pipeline of matching insertions and checking diffs was developed to determine if student submissions are correctly refactored were implemented. The pipeline depended on diffs, which are differences between two versions of code (in this case, the two versions were the student and staff solutions to exercises). Matching insertions involves removing insertions

present in both the student submission and staff solution to ignore placement agnostic code. Diff checking scans the diffs of the student submission and staff submission and checks if the changes are the same.

The students who used the tutor were somewhat successful with completing refactoring exercises. About half completed at least 11 out of the 21 exercises without too many attempts. The staff of 6.1010 also play-tested the tutor and were more successful than the students. Most of the staff was able to complete all exercises. I received feedback from the staff and some bugs were found with the exercises. However, the bugs were due to misconfigurations of the exercises. The infrastructure of the refactoring tutor allowed for the exercise author to easily make changes to the exercises to fix the bugs.

Chapter 2

Related Work

2.1 Code Smells

Code smells refer to symptoms of mediocre implementation choices that make programs hard to understand, brittle and not ready for change, and prone to bugs. Many novice programmers repeatedly introduce code smells as they program. Tufano et al. (2015), anaylzed over 500,000 commits from novice and advanced programmers and concluded that (1) both novice and advanced programmers are prone to code smells but (2) code smells from novice programmers are inherently different than those of advanced programmers. Advanced programmers tend to introduce code smells regarding software design, however novice programmers introduce lower level code smells. Code smells such as boolean laundering are more prevalent in novices than experts, for example. Izu et al. (2022) have identified certain novi code smells linked to loop constructs as they drilled deeper into the differences of novice and advanced programmers and their respective code smells. One of these code smells involves looping constructs and the idea of AskFirstOrLast. They realized that many students had conditionals in loops that check if an index has reached its final value. This is clearly a code smell, since it requires an unnecessary conditional (the range of the looping construct could simply be adjusted and the final operation could be executed before or after the loop, hence the name AskFirstOrLast). Refactoring concepts concerning conditional complexity and looping structures were addressed in several concepts of

the refactoring Tutor as they are one of the most common used structures for beginner programmers.

2.2 Why Refactoring?

As mentioned in the introduction, refactoring is crucial for the maintenance of software systems. In large scale projects, it is almost certain that code will have to be refactored. Unforeseen changes, new features, or bugs will prompt developers to refactor code. Novice programmers many times are responsible for deliberate technical debt. This includes cutting corners by writing programs with code smells to speed up development instead of thinking through general solutions that will be robust for change and easy for others to understand (Ciolkowski, 2021). In order to pass a suite of test cases, novice programmers sometimes patch their code in order to handle certain edge cases, which may end up being difficult to debug. Gaining refactoring skills at an early stage of programming will help the Student write better code and identify when code should be refactored.

2.3 Deliberate Practice

Deliberate practice makes perfect. When it comes to teaching, many studies have shown that it's not just repetition that helps improve skills, but the quality of practice. The development of skills is attained through deliberate training that drives neurological changes in the brain (Ericsson 2016). Successful musicians, athletes, scientists, etc. all have one thing in common: they all partake in purposeful practice in their field of work. This is the driving idea behind many educational programs and tools. For the Refactoring Tutor, the goal is to provide the student deliberate practice with refactoring code. With this, it is hoped that the student will benefit greatly and be capable to easily refactor code when necessary. Some of the ways this project tries to do that is by providing several curated exercises, spanning various concepts that are common across many programming languages. Since refactoring

relies on making small changes and testing if a program still works as intended, the tutor has mechanisms that ensure that the student is testing their code as they are refactoring. The tutor also attempts to provide meaningful warnings and feedback for the student. In a sense, the tutor is a resource for automatically and deliberately practicing code refactoring.

2.4 Teaching Refactoring

Teaching the student how to refactor requires them to first recognize code smells. One research study provides a resource for students to learn to identify certain code smells (Izu, 2022). The resource focused on four rules to apply when writing or reading conditional logic. They found that after explicitly teaching students the four rules to write more concise conditional statements, their code quality improved within a few weeks. They also mentioned that students who had an inadequate understanding of the programming language semantics made it difficult for students to figure out how to simplify their code. The student, therefore, must be explicitly taught about certain language semantics, while still providing practice. Although the goal for the refactoring tutor is to deliver deliberate practice, there were many opportunities to provide explanations and resources about the semantics of the programming language and how to leverage them properly to write clean code.

Keuning et al. (2021) were interested in helping novice programmers learn and practice basic refactoring skills and created a working refactor tutor that exemplifies many of the properties our tutor built upon. Their tutor had simple examples concerned with reducing conditional complexity, simplifying looping constructs, and simplifying boolean statements. Their implementation used static analyzers to evaluate whether students had made the correct changes to code that demonstrated proper refactoring. The tutor is robust, and is capable of providing clear and direct feedback to the student, however it does not have breadth in terms of refactoring concepts. The concepts were limited most to simplifying conditional statements, loops, and boolean statements. There are also only a handful of exercises which probably won't

provide the student with all the necessary tools to identify code smells and refactor programs. Additionally, the use of static analyzers to determine whether the student made the correct changes to the code is not language agnostic. Our approach attempts to further improve their tutor by expanding the different types of refactoring concepts provided while having a language agnostic way of determining whether the student's responses are correct.

Chapter 3

Code Smells

This section describes common code smells and how they are generally fixed. The code smells presented are used to create exercises in the refactoring tutor to help students how to fix and prevent writing the code smells in the future.

3.1 Boolean Laundering

Boolean laundering is defined as the unnecessary comparison between an expression which should evaluate to a boolean and a boolean constant such as `True` or `False`. It can also involve explicitly outputting boolean constants in functions, instead of using expressions that evaluate to booleans. Consider the given example:

```
# Severe case of boolean laundering
def is_even(number):
    even = number % 2 == 0
    if even == True:
        return True
    else:
        return False

# cleaner way to write the above function
```

```
10 def is_even(number):
11     return number % 2 == 0
```

The first function is riddled with boolean laundering. First of all, the conditional statement that checks if `even` is equal to `True` is unnecessary. The `even` variable itself stores `True` or `False`, depending on whether the input number is truly even. Therefore, the conditonal statement ends up having two cases. If `even` is true then, the conditional statement evaluates to `True == True`. If `even` is false, then the conditional statement evaluates to `False == True`. The explicit comparison between `even` and `True` is therefore not necessary. There is also no need to explicitly return `True` or `False`, since `even` stores the expected boolean value. Therefore, most of the code is unneeded. The second version of the function demonstrates that removing the explicit boolean check can greatly improve code readability and ease of understanding.

Almost every novice programmer has written code with boolean laundering. It is a common code smell that novice programmers don't notice, however it is easy to understand why it is unnecessary and easy to fix. Boolean laundering is one of the first code smells to learn about and practice in the refactoring tutor, since it is relatively easy to understand and checking if the student was successful in removing unnecessary boolean is less complex than checking if the student fixed other code smells.

3.2 Line too long

Having long lines of codes is generally not a good idea. When programs have long lines, it becomes hard to read. Small monitors or displays often trouble programmers since long lines require them to constantly scroll code. Even if large, wide screen monitors are available, it's not good practice to have egregiously long lines of code. Generally the solution for this code smell is simple: reduce the length of the line by breaking it up into multiple pieces. There are a couple ways to go about breaking up lines of code. One way involves creating intermediate variables to store small

expressions within a long line of code, and then using those intermediate variables to shorten the length of the long line. Consider the following two functions:

```
# return statement is too long since and hard to read
def euclidean_distance(coord1, coord2):
    '''
    coord1: (x1, y1), coord2: (x2, y2)
    '''
    return (abs(coord1[0] - coord2[0]) * abs(coord1[0] - coord2[0]) +
    ,!  abs(coord1[1] - coord2[1]) * abs(coord1[1] - coord2[1])) ** (1 /
    ,!  2)

# breaking up return statement made function much easier to understand
def euclidean_distance(coord1, coord2):
    '''
    coord1: (x1, y1), coord2: (x2, y2)
    '''
    x_dist = abs(coord1[0] - coord2[0])
    y_dist = abs(coord1[1] - coord2[1])
    return (x_dist * x_dist + y_dist * y_dist) ** (1 / 2)
```

The first function simply has a long return statement. It's hard to make sense of it due to the length of the line. The second function makes use of the two `x_dist` and `y_dist` variables to reduce the length of the return expression. The variables themselves even help document the code. Knowing that `x_dist` is the absolute value of the difference in x-coordinates, makes it easier to reason about the correctness of the return statement, which computes the euclidean distance in 2D space. This way of refactoring long lines of code is good practice, although it was difficult to check such changes in the refactoring checker.

The tutor focuses on a different way to deal with long line code smells. Specifically, long lines that have lists, tuples, or dictionaries could be split up using newlines.

In Python, expressions typically cannot be separated using newlines, however, the language does support adding newlines within lists, tuples, and dictionaries ([], (), {}). Long lines of code that have one of these types can be split in different lines. For example consider the piece of code below:

```
def euclidean_distance(coord1, coord2):
    '''
    coord1: (x1, y1), coord2: (x2, y2)
    '''
    return (
          abs(coord1[0] - coord2[0]) * abs(coord1[0] - coord2[0])
          + abs(coord1[1] - coord2[1]) * abs(coord1[1] - coord2[1])
          ) ** (1 / 2)
```

This code has not changed much from the original code, in fact, only a few newlines were inserted. However, inserting the newlines helped segment the line such that it's easier to process visually. Inserting newlines improved readability. Newlines could also be inserted to make Python dictionaries more readable. For example, it's much easier to read a newline-separated dictionary than a dictionary that has all its keys and values on a single line. Notice in the following code that the first function is a bit harder to decipher than the second function, where the keys are on different lines.

```
def reverse_audio(audio):
    '''
    Returns a new dictionary with audio data reversed.
    '''
    return {'rate': audio['rate'], 'left': audio['left'][:], 'right':
    ,'  audio['right'][:]}

def reverse_audio(audio):
    '''
```

```
    Returns a new dictionary with audio data reversed.
    '''
    return {
        'rate': audio['rate'],
        'left': audio['left'][:],
        'right': audio['right'][:]
    }
```

The refactoring tutor uses exercises that involve the student figuring out where newlines should be inserted in order to improve readability. Since identifying and inserting newlines in code is localized, it was not difficult to create a checker that made sure the student reduced the length of long lines of code.

3.3 Variable Renaming

Many novice programmers fail to choose suitable variable names for expressions. Single character variable names, generic names, or overloaded names are typically chosen instead of thinking of more expressive variable names. Debugging becomes difficult, since code with poor variable names is harder to understand. Poor variable names in code makes it harder for both the future author of the code and other developers to interact with the code, therefore, exercises that provide practice with variable renaming are included in the refactoring tutor. In the below code, the two functions calculate the compound interest earned based on some principal amount. That is fairly obvious based on the documentation. However, if the documentation weren't present, the first function would be confusing, since a developer who doesn't know about compound interest would not understand what is going on. On the other hand, if the variables have meaningful names, then a developer would have some context about how compound interest works. This is also a toy example; in large codebases, having meaningful variable names that self-document code are extremely important for understanding code, being ready for change, and preventing bugs.

```
MONTHS = 12
def compound_interest_monthly(s, r, t):
    '''
    Calculate compound interest given the starting amount, interest
    rate, and time elapsed in months.
    '''
    amount = s * ( pow( (1 + r / MONTHS), t ) )
    return amount - s

MONTHS = 12
def compound_interest_monthly(starting_amount, rate, elapsed_months):
    '''
    Calculate compound interest given the starting amount, interest
    rate, and time elapsed in months.
    '''
    amount = starting_amount * ( pow( (1 + rate / MONTHS),
    ,!  elapsed_months ) )
    return amount - starting_amount
```

3.4 Variable Naming Inconsistencies

When creating variables, it is important to be consistent to use the same style between camelCase or snake_case. Although not as common as poor variable names, having inconsistent variable names can hinder the readability of code. In the following example, variables should follow the snake_case pattern which adheres to Python variable naming conventions.

```
def average(elements):
    total_so_far, numOfElements = 0, 0

    for element in elements:
```

```
        total_so_far += element
        numOfElements += 1
    return total_so_far / numOfElements

def average(elements):
    total_so_far, num_of_elements = 0, 0

    for element in elements:
        total_so_far += element
        numOfElements += 1
    return total_so_far / num_of_elements
```

The first function named the first variable using the snake_case pattern, while naming the second variable using the camelCase pattern. The second function is consistent in using the snake_case convention for all variables.

3.5 Magic Numbers

Magic numbers are numbers in code that seem arbitrary and lack context. They are commonly found in code written by novice programmers and are not easy to understand, not ready for change, and not safe from bugs. In the following toy example, there are various magic numbers scattered across the code. These numbers may make sense to the author, however they might fail to make sense for other developers or the future author. Also, in programs where the same magic number is used more than once, refactoring becomes a hassle since all references of the same magic number would have to be changed manually. Replacing a magic number with a variable helps with three important aspects of software design. It makes the code easier to understand (ETU), ready for change (RFC), and safe from bugs (SFB).

In the first function, the magic numbers are `1.5`, `1.25`, and `3.0`. These numbers are arbitrary and should be stored as constant variables. The second function is better as the three numbers mentioned above are assigned to three different meaningful

constants that are used in the cost calculations. (Note that constants have a different naming convention than other variables. They are fully capitalized and may be separated with underscores. In the magic number exercises, the student was prompted to replace magic numbers with constants, which had the same convention.)

```
def calculate_pizza_cost(num_slices, num_toppings):
    '''
    Calculates the cost of pizza, given number of slices and number of
    toppings. Each slice has an additional cost, as well as each
    topping.
    '''
    total_slice_cost = num_slices * 1.5
    total_topping_cost = num_toppings * 1.25
    return total_slice_cost + total_topping_cost + 3.0


def calculate_pizza_cost(num_slices, num_toppings):
    '''
    Calculates the cost of pizza, given number of slices and number of
    toppings. Each slice has an additional cost, as well as each
    topping.
    '''
    BASE_COST = 3.0
    COST_PER_SLICE = 1.5
    COST_PER_TOPPING = 1.25

    total_slice_cost = num_slices * COST_PER_SLICE
    total_topping_cost = num_toppings * COST_PER_TOPPING
    return  total_slice_cost + total_topping_cost + BASE_COST
```

Magic number exercises are challenging to check, mainly because magic numbers could be created in or outside a function definition. The student is required to make

changes to code that are less localized than changes in boolean laundering exercises for example. They need to identify magic numbers, delete them, and replace them with constants, which requires a general understanding of the provided code.

3.6 Code Hoisting

The need for code hoisting is apparent when programs contain lines of code that are present in multiple branches of conditional logic. That is, if a line of code needs to be executed regardless of a conditional branch, it is still written in multiple branches. To get rid of this code smell, one must realize that the line of code must be executed regardless and then "hoist" (or pull) the line of code out of the conditional structure, while making sure the line is deleted in all the branches. In the following example, `iterations` is incremented regardless of the `(num_for_hailstone != 1)` condition, so the line, `iterations += 1`, is hoisted outside of the condition. This results in the cleaner code in the second function.

```
def get_hailstone_iterations(num_for_hailstone):
    '''
    Outputs number of times a new hailstone number is calculated until
    it reaches the terminating condition.
    '''
    iterations = 0
    while (num_for_hailstone != 1):
        if (num_for_hailstone % 2 == 0):
            num_for_hailstone /= 2
            iterations += 1
        else:
            num_for_hailstone = 3 * num_for_hailstone + 1
            iterations += 1
    return iterations

```

```
16 def get_hailstone_iterations(num_for_hailstone):
17     '''
18     Outputs number of times a new hailstone number is calculated until
19     it reaches the terminating condition.
20     '''
21     iterations = 0
22     while (num_for_hailstone != 1):
23         if (num_for_hailstone % 2 == 0):
24             num_for_hailstone /= 2
25         else:
26             num_for_hailstone = 3 * num_for_hailstone + 1
27         iterations += 1
28     return iterations
```

Code hoisting exercises require having context of the code. To know what should be changed, the student must identify which line of code is repeated and know where the line should be hoisted.

3.7 Conditional Complexity

Many times novice programmers fail to see how conditional logic can be expressed more succinctly. When constructing conditional statements, they don't have a clear understanding how control flow should work. As a simple example, the function below has an if-elif statement that could be simplified into an if-else statement. The elif part of the conditional structure is unnecessary, since `(num_for_hailstone % 2 == 0)` is the inverse of `(num_for_hailstone % 2 != 0)`. Although a minor change, it demonstrates a misunderstanding of how conditional logic should be structured.

```
1 def hailstone(num_for_hailstone):
2     sequenceOfNums = []
3     while (num_for_hailstone != 1):
```

```
        sequenceOfNums.append(num_for_hailstone)
        if (num_for_hailstone % 2 == 0):
            num_for_hailstone /= 2
        elif (num_for_hailstone % 2 != 0):
            num_for_hailstone = 3 * num_for_hailstone + 1
        sequenceOfNums.append(num_for_hailstone)
    return sequenceOfNums

def hailstone(num_for_hailstone):
    sequenceOfNums = []
    while (num_for_hailstone != 1):
        sequenceOfNums.append(num_for_hailstone)
        if (num_for_hailstone % 2 == 0):
            num_for_hailstone /= 2
        else:
            num_for_hailstone = 3 * num_for_hailstone + 1
        sequenceOfNums.append(num_for_hailstone)
    return sequenceOfNums
```

Novice programmers also fail to see when conditional structures could be collapsed. In the following example, there are two nested if-else structures. The inner if-else structure is not necessary. The conditional statements `(age < AGE_THRESHOLD)` and `is_student` could be combined into a single conditional statement. This is exactly what the second function does. Combining the two statements allows for condensing the code to use a single if-else structure. Exercises like the following were included in the refactoring tutor, since it is very common amongst novice programmers to over-complicate conditional statements. Providing such practice will help promote writing more succinct code.

```
AGE_THRESHOLD = 23
DISCOUNTED_PRICE = 10
```

```
ORIGINAL_PRICE = 15

def get_ticket_price(age, is_student):
    '''
    Gets ticket prices for a certain event given a person's
    age and student status.
    '''
    if age < AGE_THRESHOLD:
        if is_student:
            return DISCOUNTED_PRICE
        else:
            return ORIGINAL_PRICE
    else:
        return ORIGINAL_PRICE

def get_ticket_price(age, is_student):
    '''
    Gets ticket prices for a certain event given a person's
    age and student status.
    '''
    if age < AGE_THRESHOLD and is_student:
        return DISCOUNTED_PRICE
    else:
        return ORIGINAL_PRICE
```

3.8 Excessive Nesting

Complementary to simplifying conditional statements, another code smell that is common among novice programmers is excessive nesting. For example in the following code, the nesting can be reduced:

```python
def validate_and_register(user_info):
    """
    Validates and registers a user given a string input that should
    have the following form:

    'user_id,user_name'

    If the user_info doesn't have 2 entries or the user id is negative,
    then the entry is invalid and the function should return None.
    """
    parts = user_info.replace(" ", "").split(",")

    if len(parts) == 2:
        user_id = int(parts[0])
        if user_id >= 0:
            user_name = parts[1]
            return (user_id, user_name)
        else:
            return None
    else:
        return None
```

Specifically, since we return `None` if the first condition is not met, then checking if the condition evaluates to false should be used to "return fast". The same goes for the second condition. If `(user_id < 0)`, then we should return `None` immediately. There is no need to nest code in this way if we can "return fast".

```python
def validate_and_register(user_info):
    """
    Validates and registers a user given a string input that should
    have the following form:
```

```
    'user_id,user_name'

    If the user_info doesn't have 2 entries or the user id is negative,
    then the entry is invalid and the function should return None.
    """
    parts = user_info.replace(" ", "").split(",")

    if len(parts) != 2:
        return None
    user_id = int(parts[0])
    if user_id < 0:
        return None
    user_name = parts[1]
    return (user_id, user_name)
```

Notice that the nested conditions were split and the structure of the code now has the error checking in the beginning, and the main logic following the error checking. This is a trivial example, and nesting the conditional statements in this case doesn't seem terrible. However, in more complicated programs that have three or more conditions, nesting gets out of control and makes the programs more difficult to understand. This paradigm of having all of the error checking code that returns early if certain conditions are met helps prevent nesting and can greatly improve code readability and readiness to change, especially in longer and more complex programs.

Exercises that tackle code smells such as above were included in the refactoring tutor. This is one of the harder code smells to fix since it requires full understanding of the code. Major structural changes have to be made in order to separate the error checking from the main logic. However, it is particularly useful when dealing with large programs.

3.9 Lack of Idiomatic Syntax

Not having idiomatic syntax is less important than not having other code smells, however each programming language has easier ways to write different constructs, such as loops. Proficient programmers should be able to read and write each programming language's idiomatic syntax. In Python, simple loops can usually be written as list comprehensions, which encapsulate loop and conditional logic into a single line of code.

```
1  def square_numbers(numbers):
2      '''
3      Returns a new array where the numbers are squared.
4      '''
5      squared_nums = []
6      for number in numbers:
7          squared_nums.append(number ** 2)
8      return squared_nums
9
10 def squared(numbers):
11     return [number ** 2 for number in numbers]
```

The first function initializes a list to store the squared numbers, iterates through the input numbers list, appends the squares of the numbers, and outputs the `squared_nums` list. The second function has the same functionality as the first function, however the syntax is more succinct via a list comprehension.

Chapter 4

Concept Map

The exercises in the refactoring tutor were presented in a particular manner to the student. Since some code smells are harder to fix than others, the exercises were divided into groups based on concept difficulty. I created a concept map which divides the concepts into three levels. The Level 0 concept group is designed to introduce the student to the refactoring tutor. Level 1 concept groups provide the student practice with refactoring exercises that are relatively easy to refactor. The exercises in this level tend to require localized changes, not major refactoring changes. Level 2 concept groups are harder to refactor, since they require major changes to the code. Figure 4-1 depicts the concept map used for the refactoring tutor.

4.1 Level 0 Concept Group

The Level 0 concept group involves a single exercise which teaches the student how to use the refactoring tutor. This involves explaining what is expected of the student, how the prompts work, how to run test cases, how to submit an answer, and how to get hints.

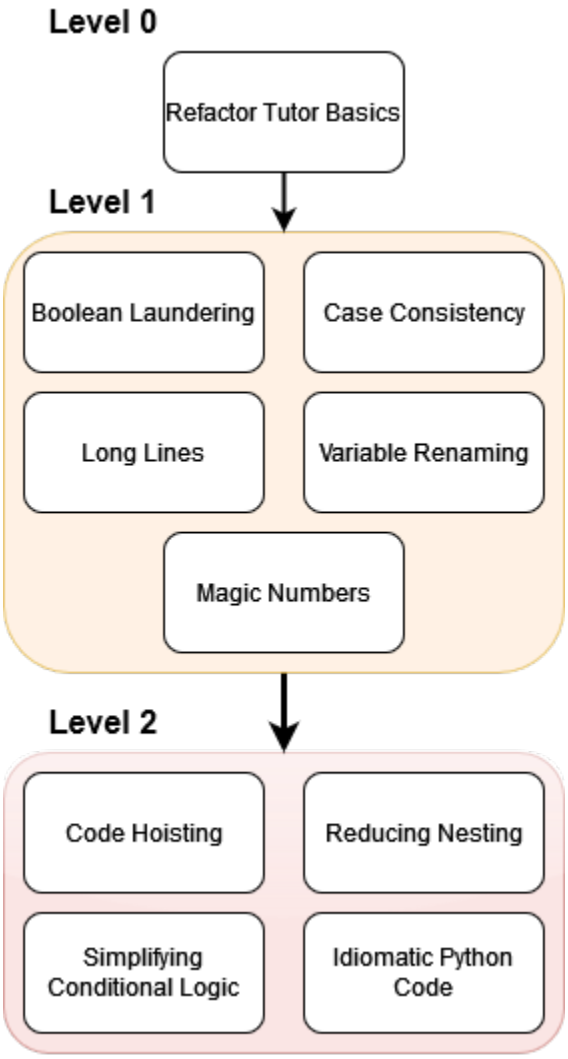

Figure 4-1: Refactoring Tutor Concept Map

The concept map of the refactoring tutor exercise groups is shown above. Level 0 consists of Refactoring tutor basics (familiarizing users with the tutor). Level 1 consists of low level refactoring techniques that are relatively easy to understand. Level 2 involves more complex refactoring techniques, with code hoisting, reducing nesting, and simplifying conditional logic being less localized than Level 1 refactoring techniques.

4.2 Level 1 Concept Groups

Level 1 concept groups include boolean laundering, case consistency, long lines, variable renaming and magic numbers. These concept groups tend to be simple to understand and master. All of the concept groups (except the Magic Numbers concept group) are mostly localized; that is students don't require contextual information

about the code to understand what to change or how to refactor code. For example, boolean laundering requires looking at a few, if not only one, lines to fix. It is very specific and localized. Long lines are also localized since students need to only find lines of code with sufficient length. Case consistency, variable renaming, and magic numbers are less localized, since students need to understand how to change variable names, however students don't need to change the structure of code in order to successfully refactor. The exercise groups mentioned above should be understood by students before they go on to tackle more complex issues. Each concept group had one or more concepts.

4.3 Level 2 Concept Groups

The more challenging refactoring concepts are found within Level 2. Exercise groups in Level 2 involve code hoisting, reducing nesting, simplifying conditional logic, and writing idiomatic python code. The exercises in these groups are not localized. They require students to gain context and understand the structure of the code they must refactor. For code hoisting, students must identify which pieces of code are executed regardless of any conditional statements. For reducing nesting and simplifying conditional logic, students must understand how to change conditional statements in order to make code succinct. Writing idiomatic python code requires understanding which code structures such as loops, conditional statements, data structures can be translated into Python's idiomatic structures. Exercises in Level 2 are designed to be completed after students master Level 1.

Chapter 5

Design

The refactoring tutor leverages the existing Praxis Tutor platform that is used to provide practice with Java and Typescript for novice programmers. The tutors made in the platform provide a series of exercises that the student completes. The refactoring tutor uses the concept map described in the previous section.

There are two types of users involved with the refactoring tutor. Exercise authors are involved in the development of exercises. They design the exercises and translate them into source files and necessary metadata. Students engage with the exercises and practice the concepts outlined in the concept map.

5.1 Exercise Authors

Creating an exercise requires creating two files. One of files contains the starting code that the student will see when starting the exercise. The other file contains the correct, refactored code that will be used to determine whether the student's submitted code is correct. The file with the starting code includes metadata in the exercise. To specify prompts, concepts, hints, and how checks will work, the author must include metadata in the starter file in the form of a YAML comment (YAML stands for *You Ain't Markup Language* and is a human-readable data-serialization language usually used for configuration files). Various exercise checkers will be responsible for using the provided metadata and the student's submitted code to determine correctness.

Figure 5-1 shows what the exercise author needs to get started making an exercise.

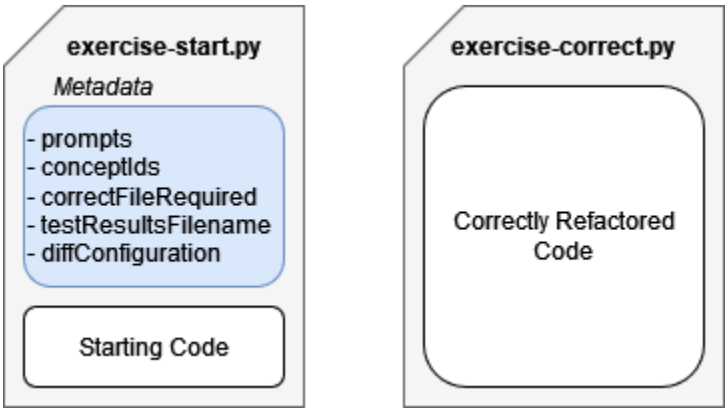

Figure 5-1: Exercise Files

Exercise metadata has configuration fields so exercise authors can specify how the exercise should work and what code will be correct. The tutor allows the exercise author to specify which lines of code should be present in the student's code, which lines of code should not be present, how long lines should be, among other configurations that will be discussed. The exercise author can create triggered hints, which are hints that are displayed to the student when a certain event or line of code is found within the student's submitted code. Triggered hints and the other metadata to configure an exercise will be discussed more in the implementation section.

5.2 The Student

The student uses the refactoring tutor within the Visual Studio Code IDE. This way, the student has access to the plethora of tools and features (syntax highlighting and an integrated terminal, for example). The refactoring tutor displays prompts for the student and serves as a guide, providing hints when the student needs assistance with the exercises.

Initially, the student is welcomed with a concept map. To get the student started with the tutor, there is an introductory exercise that explains the typical process of completing refactoring exercises; the intro exercise explains that the student must make refactoring changes to the code, run test cases, make sure they pass, and then

attempt to submit. If they need help or are stuck on an exercise, they can ask for hints by pressing the "get hint" button in the UI. The introductory exercise corresponds to the Level 0 exercise from the concept map in the previous section. After finishing Level 0 exercises, Level 1 exercises will be unlocked in the concept map. Once a student fully completes the exercises in a concept group, a green check mark will appear next to the completed concept group. Figure 5-2 shows the initial view of the refactoring tutor. As can be seen, the tutor is integrated with the Visual Studio Code IDE. If the student wishes to attempt an exercise group again, they can do so by navigating to the concept group and clicking it again.

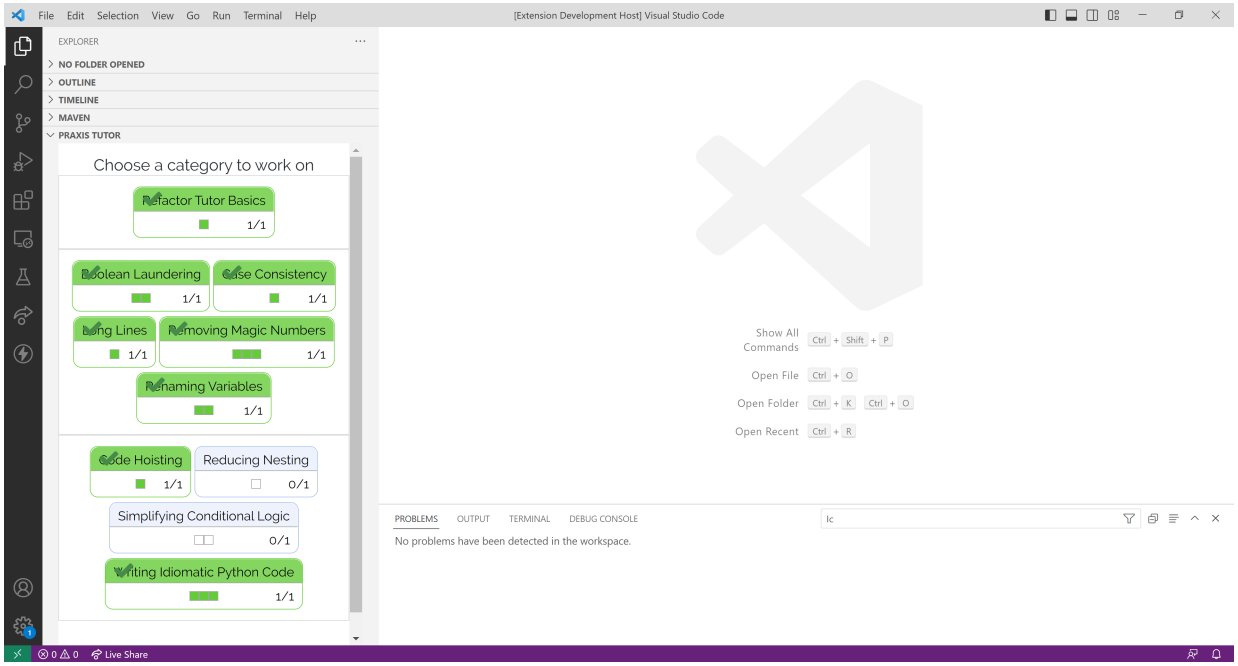

Figure 5-2: Refactoring Tutor Concept Map Progress

When a student clicks on an exercise group, one of the exercises from the group is displayed to the student. The tutor then displays a prompt to the student while the VSCode IDE displays the code associated with the prompt. Figure 5-3 shows an exercise from the Code Hoisting exercise group. There is a prompt on the left hand side of the explorer of the IDE and code for the student to refactor on the right hand side. In general, the prompts will require the student to either delete, reorganize, or add certain lines of code. An important aspect to refactoring that is emphasized by

the tutor is that refactoring itself should not affect functionality of the provided code. Therefore, most exercises have test cases, or a series of inputs and expected outputs that the code must produce. The student must run the test cases from the IDE.

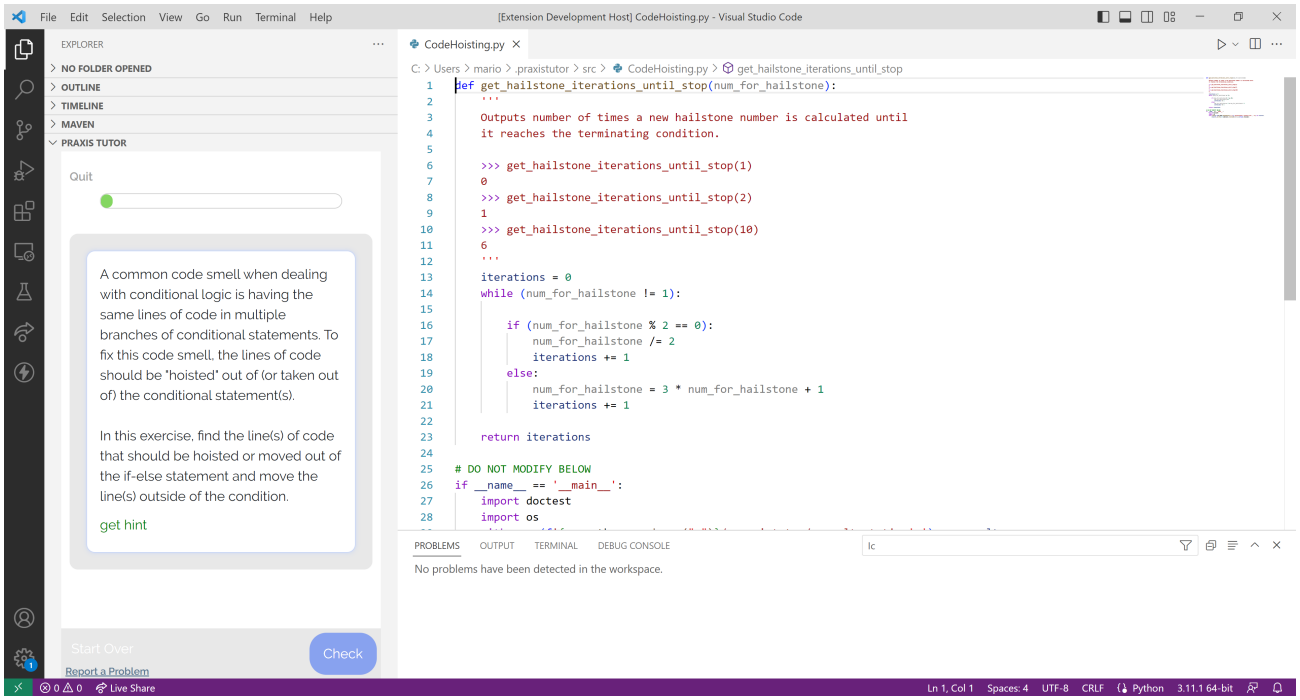

Figure 5-3: Code Hoisting Exercise

Automatically running test cases was considered, however, promoting students to run tests cases themselves, even if it simply requires a press of a button, is another important goal of the tutor. If students develop a habit of running test cases whenever they change code, it will hopefully help them catch mistakes early on. For example, if the student has a program that requires major refactoring and the student only runs the test cases when they have made significant changes to the program, then it is possible the changes introduced bugs. The student will then have to debug them and figure out which changes cause the test cases to fail, which will take time. On the other hand, if the student continually runs test cases while refactoring the program, then they will catch the issue immediately. This is good practice, and even when implementing new features, tests should be run to ensure correct functionality. Therefore, requiring test cases to be run is intentional, even when the student makes small changes to the code in the exercise.

When test cases are run in the IDE, the terminal will show whether tests pass or not. In Figure 5-4, the test cases have been run; the terminal at the bottom of the figure will let the student know if the tests have passed. If the student does not run the test cases and attempts to submit the exercise, then an error message will be displayed on the tutor as can be seen in Figure 5-5. In the tutor side, a message in red text appears, notifying the student that test cases should be run before submitting the exercise. If the student fails the test cases and attempts to submit, then an error message will appear on the tutor side and the student will be unable to submit the exercise. The refactoring tutor will have a sizable reliance on test cases to make sure students successfully refactor the provided code, but also do not change functionality.

Even if test cases pass, the student must still check whether the refactored code meets the prompt's requirements. If not, then the tutor will be outlined with red, signifying to the student that the submitted code was not refactored correctly. If the student does successfully pass the test cases and submits correctly refactored code, then the tutor will be outlined with green, signifying completion. The student will be able to move on the next exercise in the exercise group. Figures 5-6 and 5-7 show what the student will see when either failing or completing the exercise.

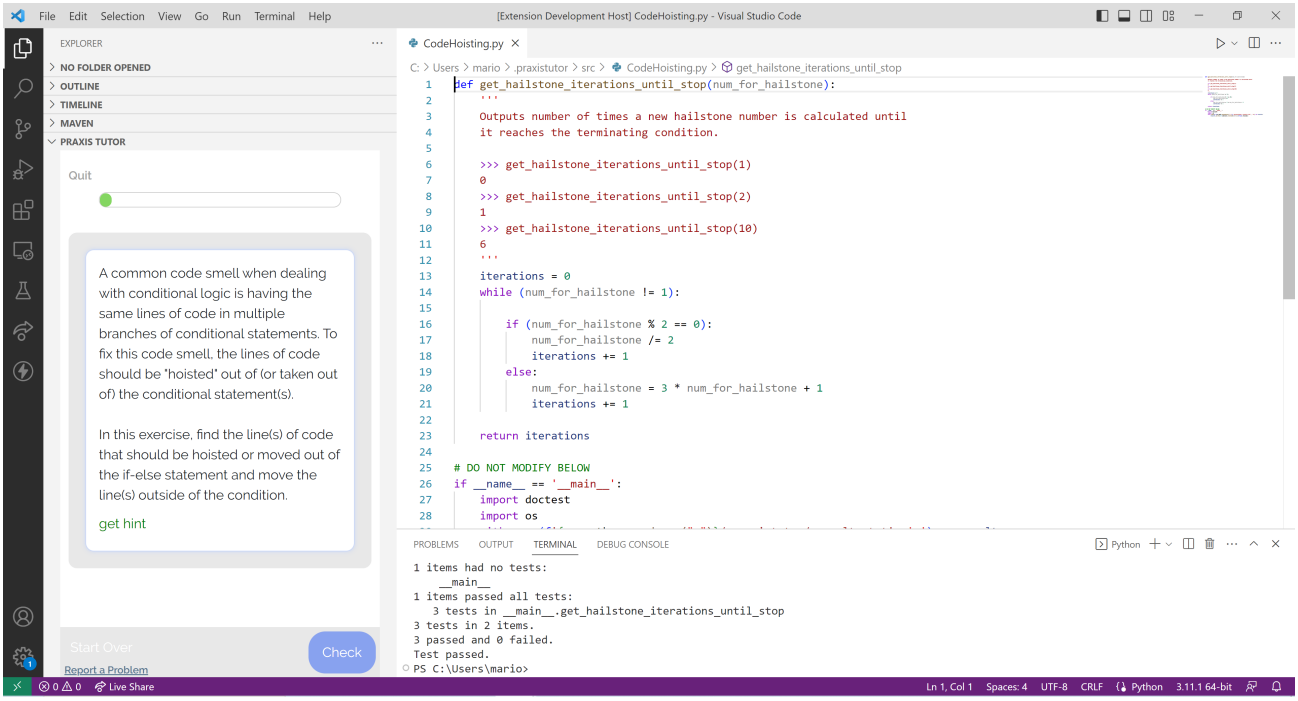

Figure 5-4: Running Test Cases on Refactoring Tutor

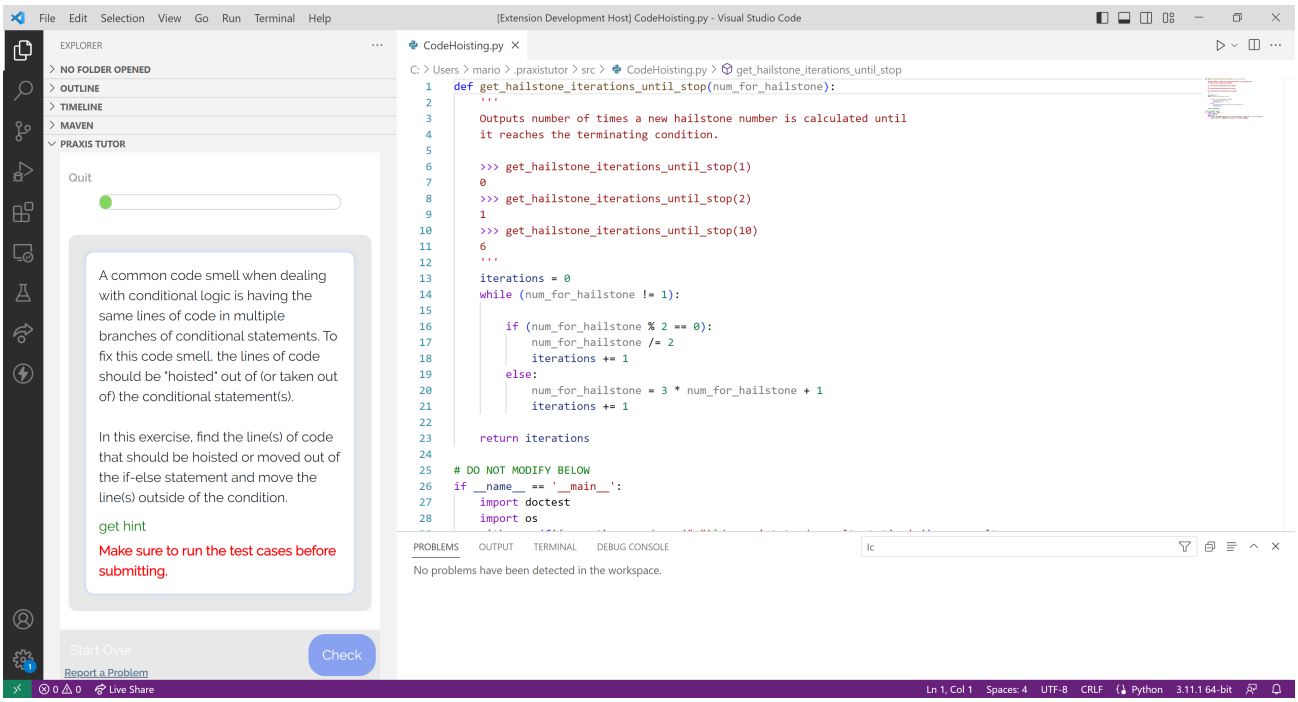

Figure 5-5: Submitting Exercise before running Test Cases