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

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

List of Figures

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 introductory 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:

```python
# 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
```

```python
def is_even(number):
    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:

```python
# 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:

```python
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.

```python
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.

```python
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.

```python
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.)

```python
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.

```python
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

```

```python
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
        else:
            num_for_hailstone = 3 * num_for_hailstone + 1
        iterations += 1
    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.

```python
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
        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.

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

```python
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.

```python
def square_numbers(numbers):
    '''
    Returns a new array where the numbers are squared.
    '''
    squared_nums = []
    for number in numbers:
        squared_nums.append(number ** 2)
    return squared_nums

def squared(numbers):
    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.

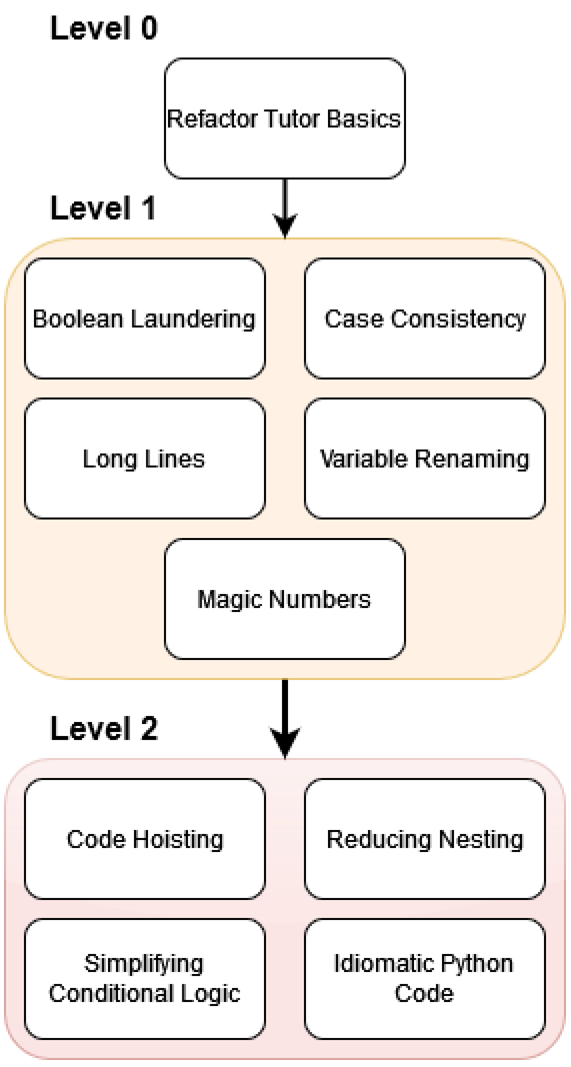


Figure 4-1: Refactoring Tutor Concept Map

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.

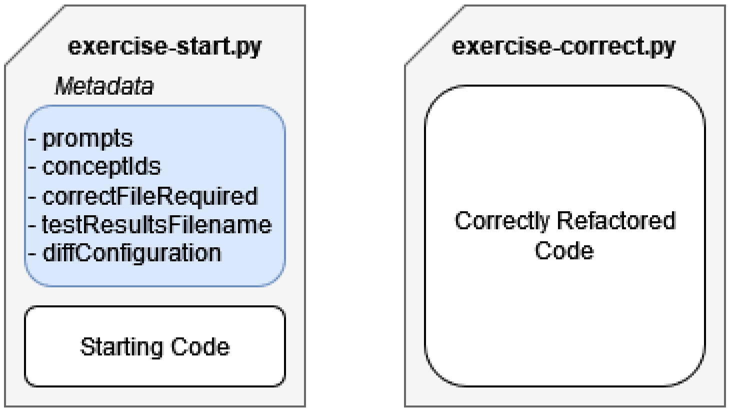


Figure 5-1: Exercise Files

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.

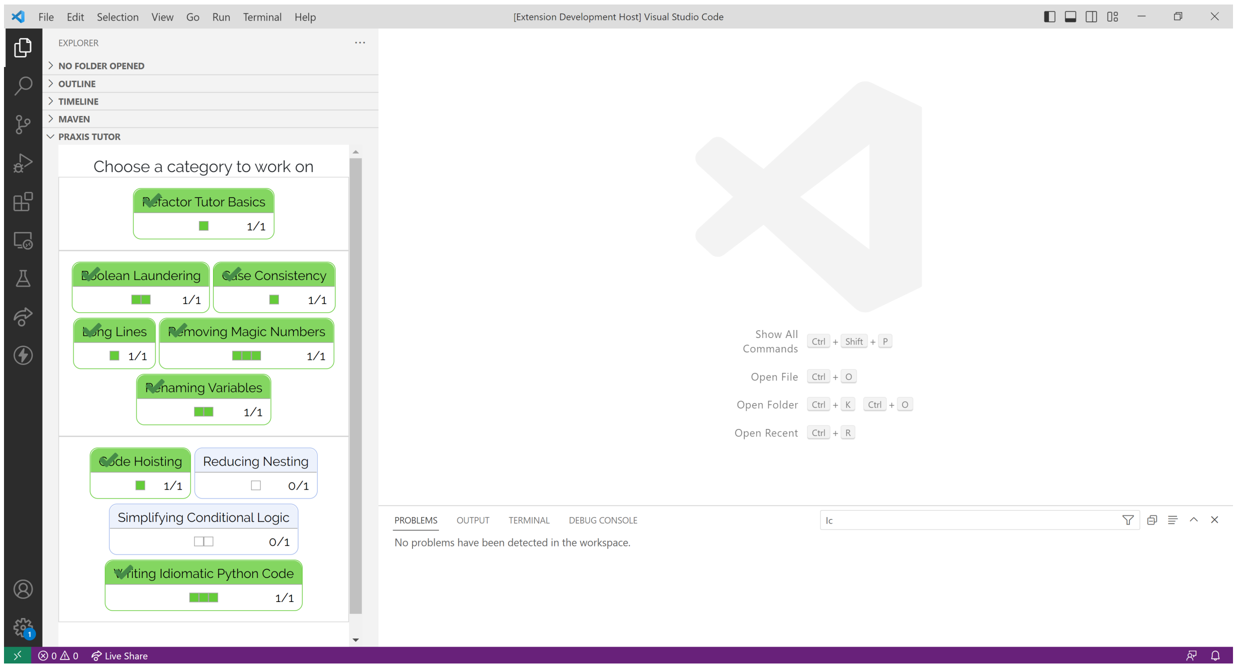
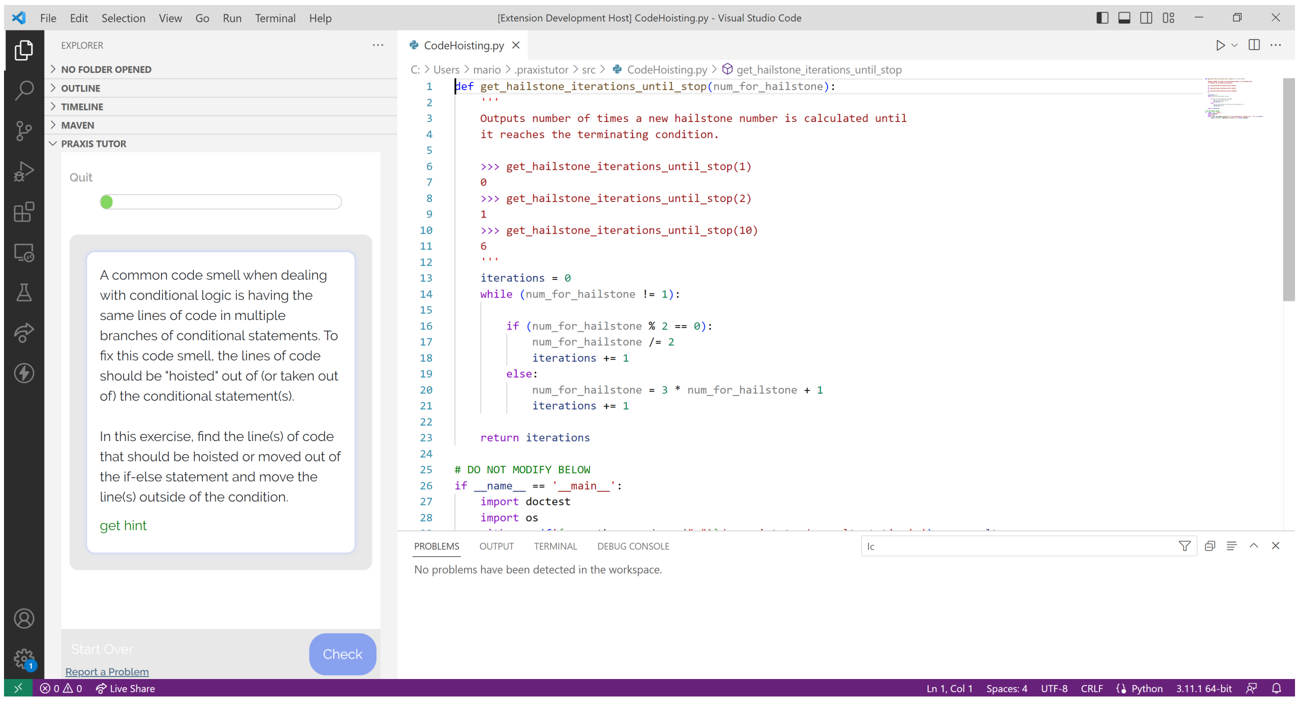

File Edit Selection View Go Run Terminal Help
[Extension Development Host] Visual Studio Code
EXPLORER
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PRAXIS TUTOR
Choose a category to work on
Refactor Tutor Basics 1/1
Boolean Laundering 1/1
Case Consistency 1/1
Long Lines 1/1
Removing Magic Numbers 1/1
Renaming Variables 1/1
Code Hoisting 1/1
Reducing Nesting 0/1
Simplifying Conditional Logic 0/1
Writing Idiomatic Python Code 1/1
Show All Commands Ctrl + Shift + P
Open File Ctrl + O
Open Folder Ctrl + K Ctrl + O
Open Recent Ctrl + R
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
No problems have been detected in the workspace.
Live Share

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.


File Edit Selection View Go Run Terminal Help
[Extension Development Host] CodeHoisting.py - Visual Studio Code
EXPLORER
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PRAXIS TUTOR
Quit
CodeHoisting.py
C: > Users > mario > .praxistutor > src > CodeHoisting.py > get_hailstone_iterations_until_stop
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Start Over
Check
Report a Problem
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.11.1 64-bit
Live Share

A common code smell when dealing with conditional logic is having the same lines of code in multiple branches of conditional statements. To fix this code smell, the lines of code should be "hoisted" out of (or taken out of) the conditional statement(s).

In this exercise, find the line(s) of code that should be hoisted or moved out of the if-else statement and move the line(s) outside of the condition.

get hint

```python
def get_hailstone_iterations_until_stop(num_for_hailstone):
    '''
    Outputs number of times a new hailstone number is calculated until
    it reaches the terminating condition.

    >>> get_hailstone_iterations_until_stop(1)
    0
    >>> get_hailstone_iterations_until_stop(2)
    1
    >>> get_hailstone_iterations_until_stop(10)
    6
    '''
    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

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
```

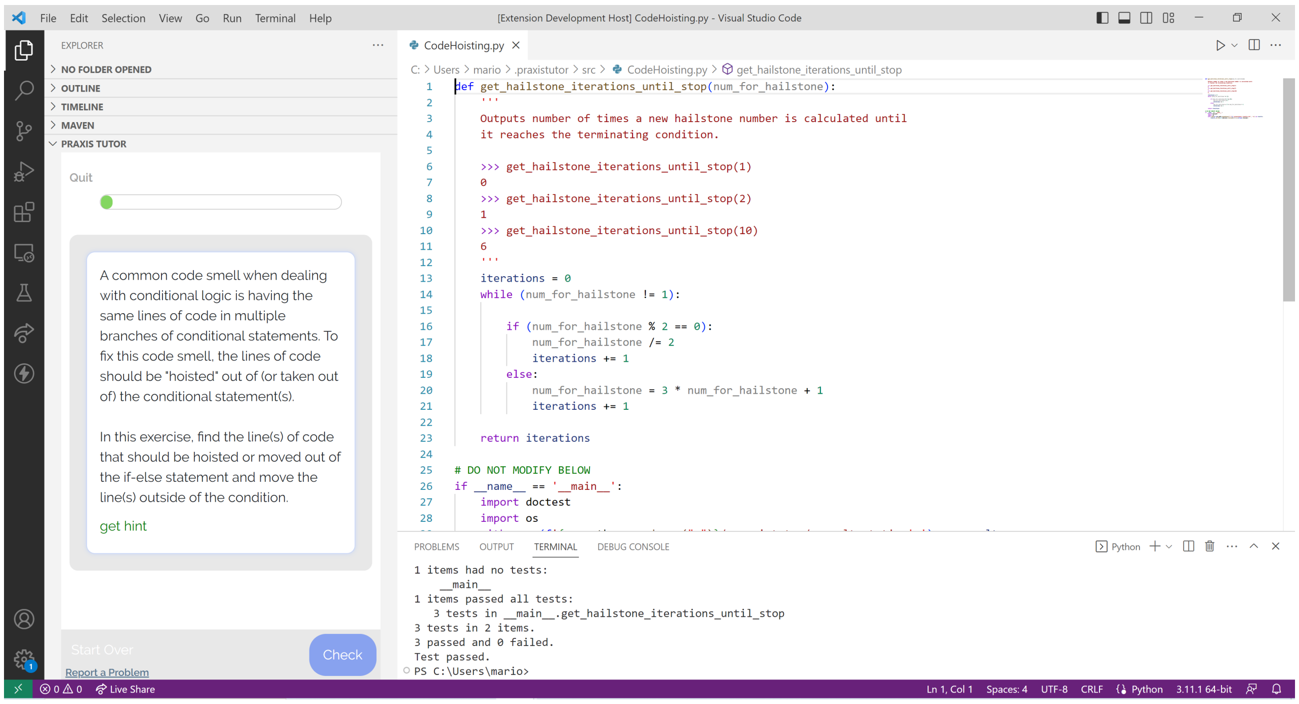
No problems have been detected in the workspace.

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.


File Edit Selection View Go Run Terminal Help
[Extension Development Host] CodeHoisting.py - Visual Studio Code
EXPLORER
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PRAXIS TUTOR
Quit
CodeHoisting.py
C: > Users > mario > .praxistutor > src > CodeHoisting.py > get_hailstone_iterations_until_stop
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Python
Start Over
Check
Report a Problem
Live Share
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.11.1 64-bit

A common code smell when dealing with conditional logic is having the same lines of code in multiple branches of conditional statements. To fix this code smell, the lines of code should be "hoisted" out of (or taken out of) the conditional statement(s).

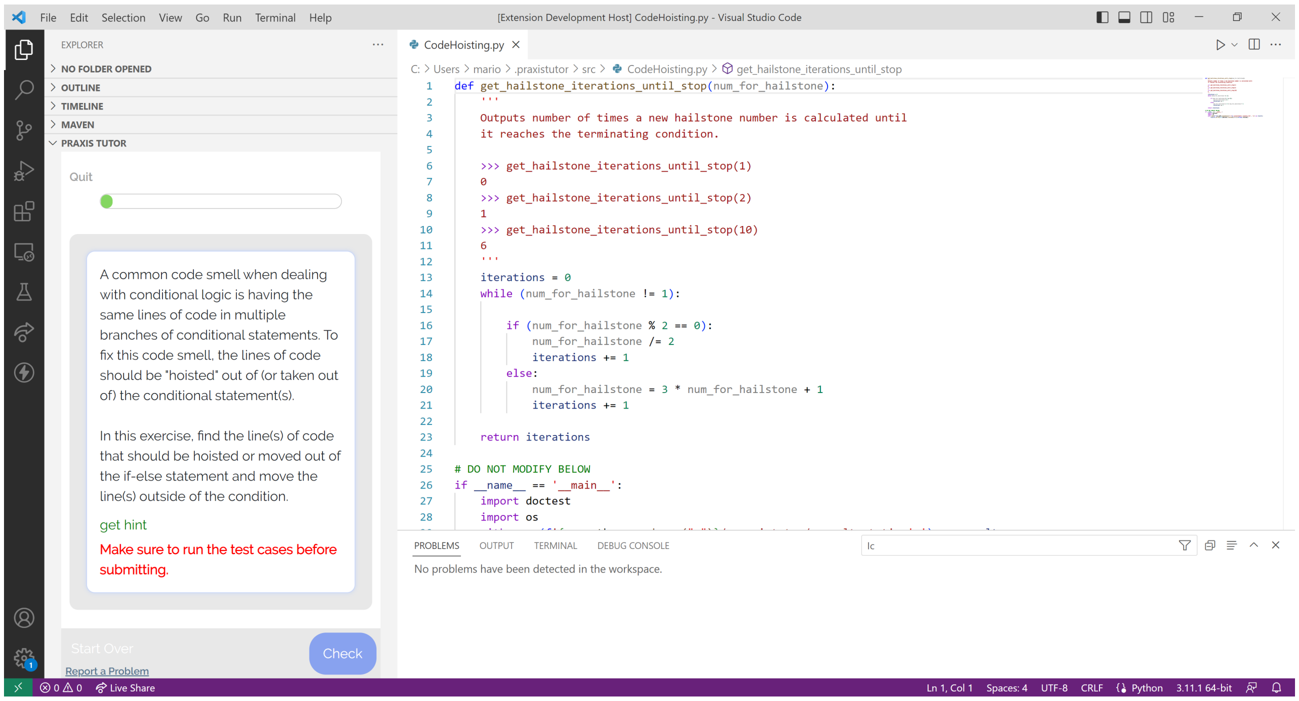
In this exercise, find the line(s) of code that should be hoisted or moved out of the if-else statement and move the line(s) outside of the condition.

get hint

```python
def get_hailstone_iterations_until_stop(num_for_hailstone):
    '''
    Outputs number of times a new hailstone number is calculated until
    it reaches the terminating condition.

    >>> get_hailstone_iterations_until_stop(1)
    0
    >>> get_hailstone_iterations_until_stop(2)
    1
    >>> get_hailstone_iterations_until_stop(10)
    6
    '''
    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

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
```

```
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.get_hailstone_iterations_until_stop
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
PS C:\Users\mario>
```

Figure 5-4: Running Test Cases on Refactoring Tutor


File Edit Selection View Go Run Terminal Help
[Extension Development Host] CodeHoisting.py - Visual Studio Code
EXPLORER
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PRAXIS TUTOR
Quit
CodeHoisting.py
C: > Users > mario > .praxistutor > src > CodeHoisting.py > get_hailstone_iterations_until_stop
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Start Over
Check
Report a Problem
Live Share
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.11.1 64-bit

A common code smell when dealing with conditional logic is having the same lines of code in multiple branches of conditional statements. To fix this code smell, the lines of code should be "hoisted" out of (or taken out of) the conditional statement(s).

In this exercise, find the line(s) of code that should be hoisted or moved out of the if-else statement and move the line(s) outside of the condition.

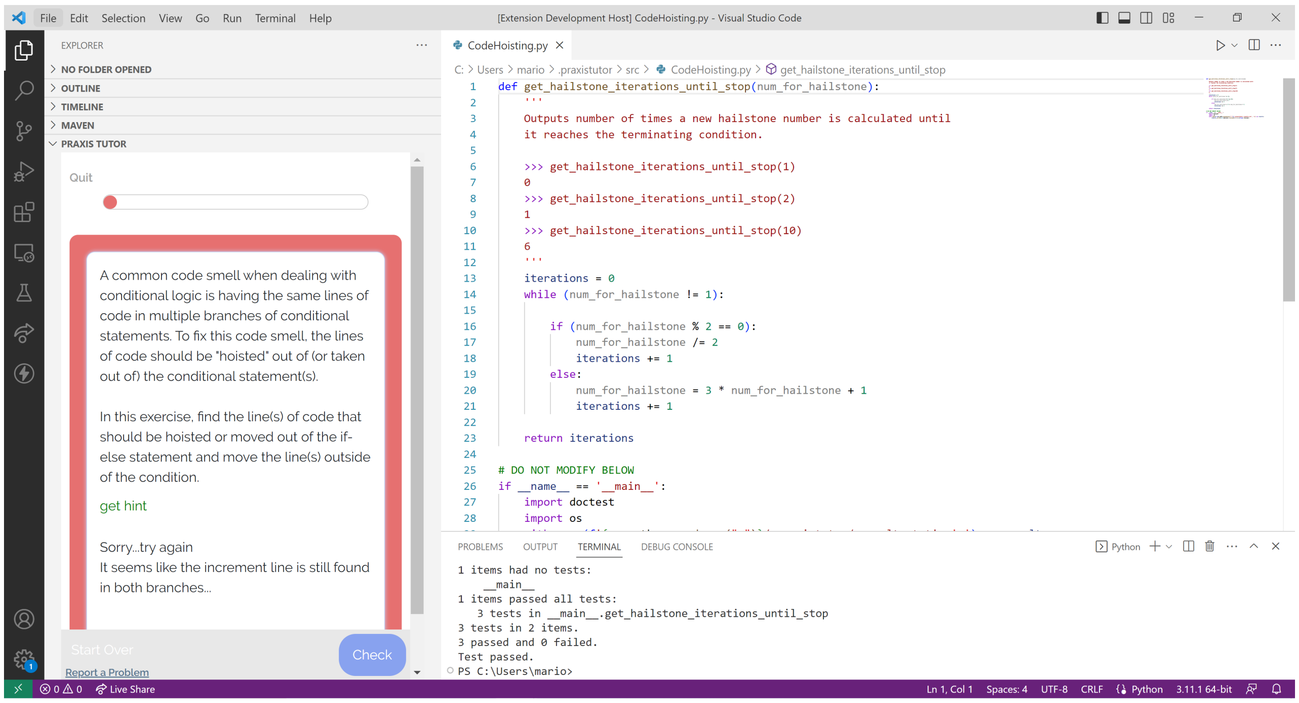
get hint

Make sure to run the test cases before submitting.

```python
def get_hailstone_iterations_until_stop(num_for_hailstone):
    '''
    Outputs number of times a new hailstone number is calculated until
    it reaches the terminating condition.

    >>> get_hailstone_iterations_until_stop(1)
    0
    >>> get_hailstone_iterations_until_stop(2)
    1
    >>> get_hailstone_iterations_until_stop(10)
    6
    '''
    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

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
```

```
No problems have been detected in the workspace.
```

Figure 5-5: Submitting Exercise before running Test Cases


File Edit Selection View Go Run Terminal Help
[Extension Development Host] CodeHoisting.py - Visual Studio Code
EXPLORER
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PRAXIS TUTOR
Quit
CodeHoisting.py
C: > Users > mario > .praxistutor > src > CodeHoisting.py > get_hailstone_iterations_until_stop
Start Over
Check
Report a Problem
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Python
Live Share
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.11.1 64-bit

A common code smell when dealing with conditional logic is having the same lines of code in multiple branches of conditional statements. To fix this code smell, the lines of code should be "hoisted" out of (or taken out of) the conditional statement(s).

In this exercise, find the line(s) of code that should be hoisted or moved out of the if-else statement and move the line(s) outside of the condition.

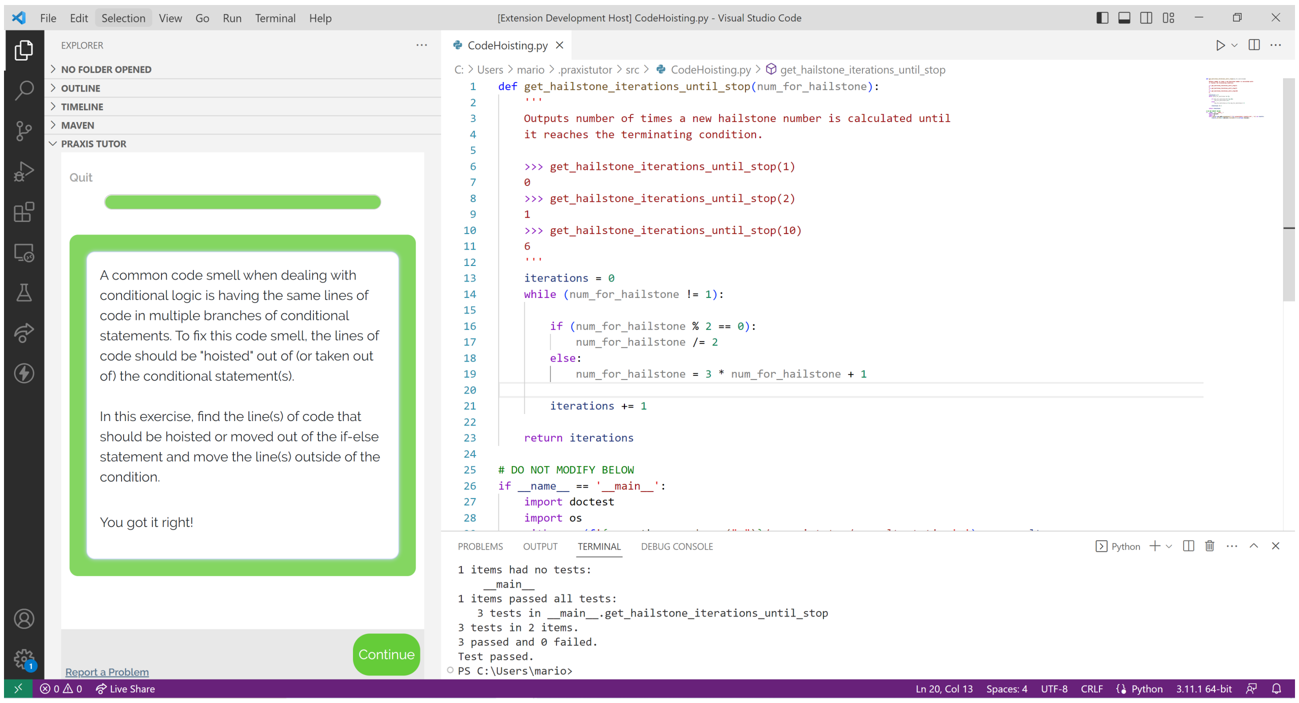
get hint

Sorry...try again
It seems like the increment line is still found in both branches...

```python
def get_hailstone_iterations_until_stop(num_for_hailstone):
    '''
    Outputs number of times a new hailstone number is calculated until
    it reaches the terminating condition.

    >>> get_hailstone_iterations_until_stop(1)
    0
    >>> get_hailstone_iterations_until_stop(2)
    1
    >>> get_hailstone_iterations_until_stop(10)
    6
    '''
    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

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
```

```
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.get_hailstone_iterations_until_stop
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
PS C:\Users\mario>
```

Figure 5-6: Failing Exercise on Refactoring Tutor


File Edit Selection View Go Run Terminal Help
[Extension Development Host] CodeHoisting.py - Visual Studio Code
EXPLORER
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PRAXIS TUTOR
Quit
CodeHoisting.py
C: > Users > mario > .praxistutor > src > CodeHoisting.py > get_hailstone_iterations_until_stop
Continue
Report a Problem
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Python
Live Share
Ln 20, Col 13 Spaces: 4 UTF-8 CRLF Python 3.11.1 64-bit

A common code smell when dealing with conditional logic is having the same lines of code in multiple branches of conditional statements. To fix this code smell, the lines of code should be "hoisted" out of (or taken out of) the conditional statement(s).

In this exercise, find the line(s) of code that should be hoisted or moved out of the if-else statement and move the line(s) outside of the condition.

You got it right!

```python
def get_hailstone_iterations_until_stop(num_for_hailstone):
    '''
    Outputs number of times a new hailstone number is calculated until
    it reaches the terminating condition.

    >>> get_hailstone_iterations_until_stop(1)
    0
    >>> get_hailstone_iterations_until_stop(2)
    1
    >>> get_hailstone_iterations_until_stop(10)
    6
    '''
    iterations = 0
    while (num_for_hailstone != 1):

        if (num_for_hailstone % 2 == 0):
            num_for_hailstone /= 2
        else:
            num_for_hailstone = 3 * num_for_hailstone + 1

        iterations += 1

    return iterations

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
```

```
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.get_hailstone_iterations_until_stop
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
PS C:\Users\mario>
```

Figure 5-7: Completing Exercise on Refactoring Tutor

Chapter 6

Implementation

As mentioned in the Design section, the refactoring tutor uses the Praxis Tutor infrastructure, which is comprised of three main components: the Mongo Database (MongoDB), the VSCode Extension, and the Tutor Web Application. MongoDB is used to store relevant refactoring exercises ordered by concepts. The exercises are stored as source files which contain both the code the student will view in the VSCode IDE and the metadata for the exercise in the form of a YAML comment. The YAML comment is what the exercise author modifies to include prompts, create hints, and specify the types of validation checks required for the exercise. The VSCode extension is a small plugin that allows interfacing with the IDE. The student views exercises on the IDE and edits code as they wish. The Web Application (Web App) is the workhorse of the tutor, since it is responsible for fetching exercises, checking the student's submissions, and providing useful feedback. It has to communicate with MongoDB and the VSCode Extension as can be seen in Figure 6-1. The architecture of the Refactoring Tutor and the typical workflow that is expected of the student is shown in Figure 6-1. There were a few changes to the Praxis Tutor infrastructure (which will be explained in the next sections) to allow for robustly checking refactoring exercises, however, the core architecture remains the same.

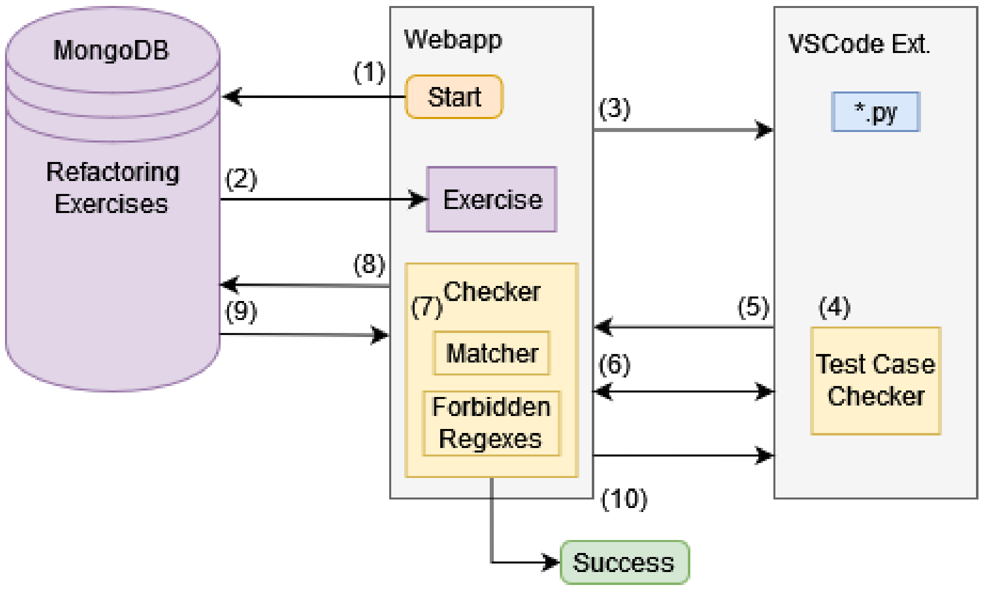


Figure 6-1: Refactoring Tutor Architecture

(1): The student starts the refactoring tutor and clicks on an available exercise.
(2): MongoDB sends back desired exercise to Web Application.
(3): Exercise sent from Web App to VSCode Plugin. Exercise is set up for use by the student.
(4): The student requests to check answer. The Extension first makes sure that test cases (doctests) have been run on the latest version of the student code before sending request to the Web App.
(5): Request to check answer is sent to Web App.
(6): Web App asks Extension for student answer; extension sends the student answer to Web App.
(7): Checker tests for constraints and rules set by instructor or exercise author.
(8): Web App asks MongoDB to provide staff solution.
(9): Check if the student answer conforms to the constraints and rules in the staff solution. Triggered hints provide guidance, if any.
(10): If the Student's answer is correct, then tutor will display a successful message. If not, the student will see a message to try again.

6.1 Exercise Checking

Refactoring is a subjective process and there exist multiple ways to refactor code. There are better ways to refactor pieces of code than others, however the space of valid ways to refactor code is very large, which makes checking refactoring exercises extremely difficult. It is much harder to check if code was properly refactored than checking for correctness. Hence, checking for refactoring has proven the most difficult aspect of this project. To make checking refactoring easier, every refactoring concept was tested individually. That is, each exercise checks for a specific refactoring technique. Exercises with multiple refactoring techniques would make correctness checking much harder; the space of solutions would be even larger. Scoping the exercises to only a single refactoring technique has proven useful and the current methods for checking correctness for the exercises explained in the concept map have been robust so far.

In order to understand how the main method for checking, the notion of diffs must be understood. Diffs refer to the differences of two versions of text. Diffs are used in version control systems such as git as it provides a way to show the differences between two versions of the same file. As an example, suppose there are two versions of a file: **Version A** and **Version B**. A diff given these two versions will determine their differences. In general, it will provide data on insertions, deletions, and unchanged text. This is useful in order to determine where and what changed from **Version A** to **Version B**.

Diffs are an integral aspect of the refactoring tutor's correctness checkers. They are used to help determine the differences between the student's code and the correctly refactored code specified by the exercise author. A simple use case for diffs in boolean laundering exercises, for example, is determining whether the student deleted the boolean values `True` or `False`. Given that a diff contains information about deleted text, it is trivial to check for correctness. For example, the code below can be refactored by removing the `== True` part of the code in line 10 of the code shown below.

```python
def is_even(number):
    """ Return true if number is even, false otherwise. """
    return number % 2 == 0

def sum_evens(sequence):
    """ Return the sum of even numbers in sequence list.
    Assume all numbers non-negative. """
    even_sum = 0
    for number in sequence:
        if is_even(number) == True:
            even_sum += number
    return even_sum
```

The resulting diff will be a sequence of unchanged, and deleted text (there are no insertions in this case). The deletion might be expressed as the following:

```
{"Type": Deletion, Line: 10, "Text": "== True"}
```

When using diffs, the granularity can be chosen. The granularity can be adjusted to single characters, words (sequence of non-space, non-newline), or whole lines. The diffing used in this design uses word granularity in diffs. It was found that using word granularity works better for refactoring exercises, since usually the refactoring exercises look for changes in words.

6.2 Diff Checking

Checking the student's answer for an exercise requires a staff solution. A staff solution is one correctly refactored version of the exercise. The goal of the refactor tutor is to guide the student into making a series of changes such that their code gets as close to the staff solution as possible. Simply checking if the raw text of the student solution is the same as the raw text of the staff solution is a naive way to check for correctness. This method would not be robust to irrelevant deviations of the student solution from the staff solution, and would not be able to provide proper hints to the student since there would be no concept of code insertions and deletions. Using diffs is a more robust method to determine correctness. However, this method requires three versions of the exercise code: the student code, staff code and original code. The original exercise code is used to create two distinct diffs. The diff between the original exercise code and the student code will determine what the student left unchanged, deleted, or inserted while doing the exercise. This diff will be referred to as the student diff. The diff between the original exercise code and the staff code describes which parts of the code should be left unchanged, which lines should be deleted, or what should be inserted after refactoring. This will be referred to as the staff diff. Once both diffs are computed, the exercise can be checked for correctness. By sequentially scanning both diffs and ensuring that each contains the same unchanged, deleted, or inserted pieces of code, the exercise can be checked. If the student diff does not have the correct deletions or insertions, the exercise will be marked wrong, but they may receive feedback depending on the exercise author's configuration. If the student diff does not have the same unchanged code, the exercise will also be marked wrong, and feedback will similarly be provided to the student. The student will be informed of changes that weren't supposed to be made. Figure 6-2 depicts the process of creating the student and staff diffs and comparing them to determine correctness for a given exercise.

Diff checking does work for exercises focusing on localized changes to code. This includes the boolean laundering and case consistency exercises. Some simplifying conditional exercises only need diff checking also.

Checking for identical diffs alone though, is not robust for all types of exercises. Namely, exercises that should have tolerance to student solutions such as those providing practice with magic numbers, renaming variable names, reducing conditional complexity, etc. should allow for sensible answers that differ from the staff solution. For example, when magic numbers are removed, a constant should be created to replace the magic number. Generally, there are multiple sensible places in code where a constant could be defined. The student may not define the constant in the same place where it was defined in the staff solution, however it can still be perfectly fine. Diff checking will not be robust to these scenarios.

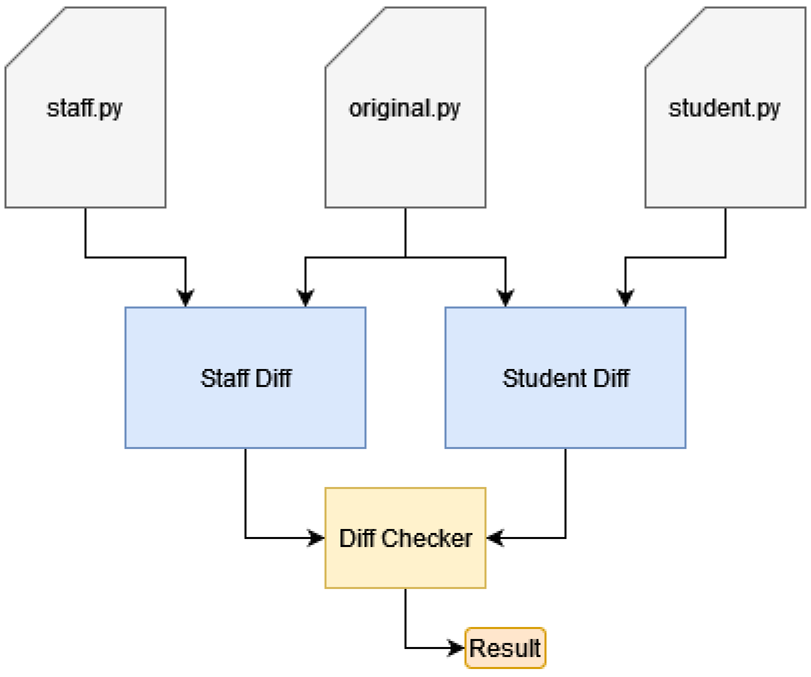
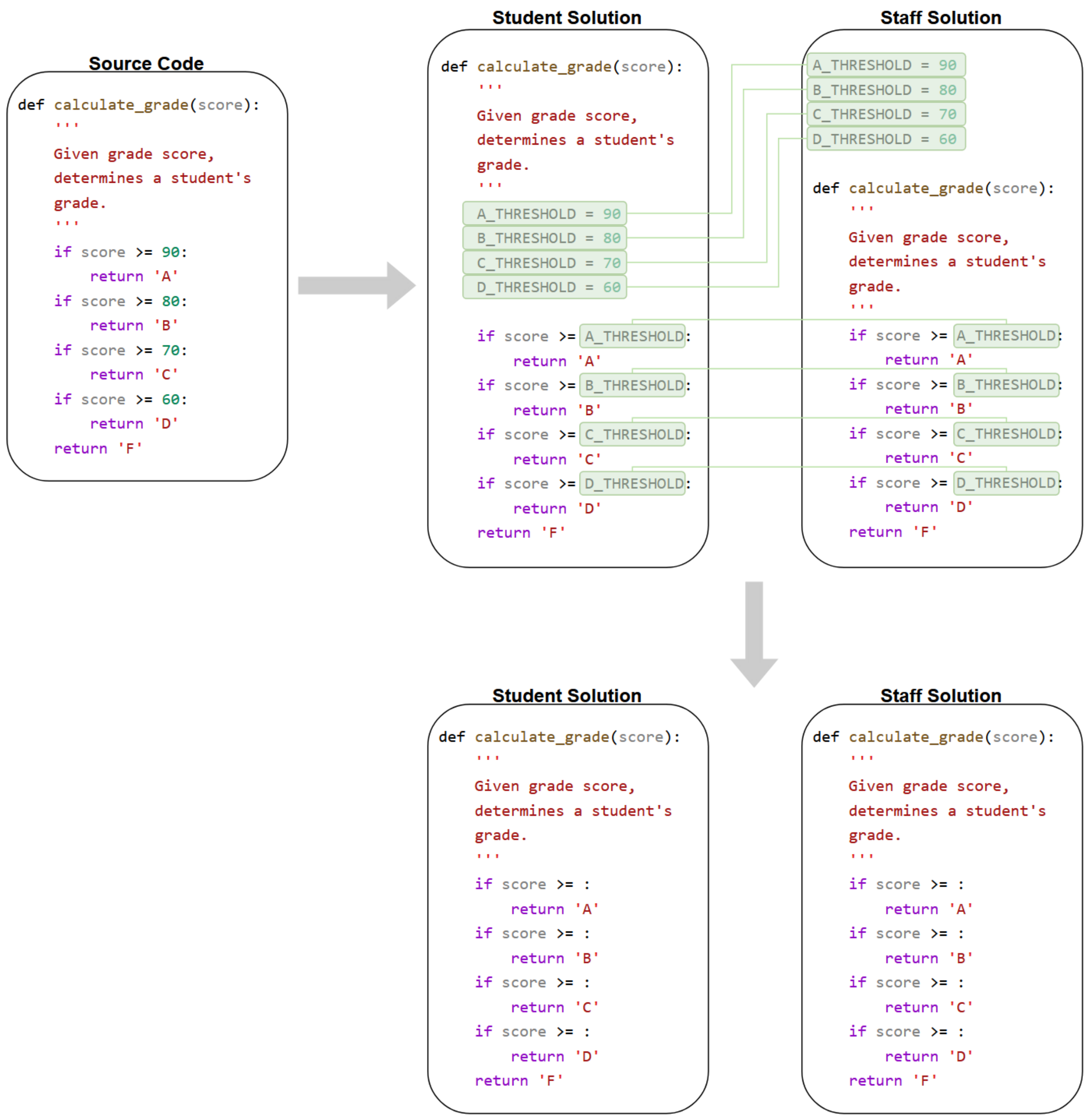


Figure 6-2: Diff Checker

There are three different files: the original source code, the staff solution, and the student solution. A diff between the staff solution and the original source code will have to be compared against a diff between the student solution and the original source code in order to determine if the student correctly refactored the prompted code.

6.3 Diff Matching

Increasing tolerance for exercises that allow multiple sensible solutions requires another process for handling diffs. Such a process will be referred to as Diff Matching, which is executed before running a diff check on the student and staff diffs. Instead of only scanning through diffs and checking for equality as was described above, diff matching involves scanning two diffs and matching insertions that are found in both diffs. Matching a pair of insertions means that both insertions are found in the diffs, regardless of order. If a pair of insertions is matched, then they are removed from the diffs. The idea is that matching all pairs of insertions in the student and staff diffs, removes lines of code that are placement agnostic and are found in both the student and staff diffs. For example, in Figure 6-3, `A_THRESHOLD = 90` is found in both the student and staff diffs. In the student diff, `A_THRESHOLD = 90` is found within the `calculated_grade()` function, whereas in the staff diff, the line is found outside of the function. In both of these cases, the magic number is removed and replaced with a valid constant; the student solution is correct. Therefore, in order for the diff checking process to work, the same insertions must be matched and removed from both the student and staff diffs.



Source Code

```python
def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    if score >= 90:
        return 'A'
    if score >= 80:
        return 'B'
    if score >= 70:
        return 'C'
    if score >= 60:
        return 'D'
    return 'F'
```

Student Solution

```python
def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    A_THRESHOLD = 90
    B_THRESHOLD = 80
    C_THRESHOLD = 70
    D_THRESHOLD = 60

    if score >= A_THRESHOLD:
        return 'A'
    if score >= B_THRESHOLD:
        return 'B'
    if score >= C_THRESHOLD:
        return 'C'
    if score >= D_THRESHOLD:
        return 'D'
    return 'F'
```

Staff Solution

```python
A_THRESHOLD = 90
B_THRESHOLD = 80
C_THRESHOLD = 70
D_THRESHOLD = 60

def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    if score >= A_THRESHOLD:
        return 'A'
    if score >= B_THRESHOLD:
        return 'B'
    if score >= C_THRESHOLD:
        return 'C'
    if score >= D_THRESHOLD:
        return 'D'
    return 'F'
```

Student Solution

```python
def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    if score >= :
        return 'A'
    if score >= :
        return 'B'
    if score >= :
        return 'C'
    if score >= :
        return 'D'
    return 'F'
```

Staff Solution

```python
def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    if score >= :
        return 'A'
    if score >= :
        return 'B'
    if score >= :
        return 'C'
    if score >= :
        return 'D'
    return 'F'
```

Figure 6-3: Matching Student And Staff Diffs

The process of diff matching is shown above. The diffs for the student and staff solutions are computed. For demonstration purposes, only the insertions in the diffs are depicted by the green boxes. The lines between the green boxes represent matching two insertions between the two diffs. For example, `A_THRESHOLD = 90` in the student solution is matched to the same text in the staff solution. Matching all the insertions between the student and staff diffs results in removing them from the diffs. After removing the matched insertions, the diffs represent the code seen in the bottom two boxes.

6.4 Exercise Metadata

As mentioned in the previous sections, the metadata stored in the YAML comments of exercises specifies the prompts, conceptIds, and the requirements for the Web Application to mark the student's solution to exercises as correct. To support refactoring exercises, several fields were added to the Praxis Tutor metadata model. An example of the metadata for a refactoring exercise is shown below:

```
#<yaml>
#  - id: Magic-numbers
#    conceptIds:
#      - python::removing-single-magic-number
#    prompts:
#      - >-
#          <p>Magic numbers are numbers that may not seem random to you,
#          but do seem random to others who end up reading your code.
#          It is bad practice to introduce numbers other than the most
#          common numbers used while coding (0, 1, 2 for example)
#          without adding a comment or creating a constant with a
#          descriptive name. In this exercise we'll want to do the
#          latter.<br>
#
#          <p>Find and remove the magic number in the following code.
#          Replace the magic number with a constant variable with name,
#          <b>MINIMUM_PASSING_SCORE</b>.</p>
#      - >-
#          https://py.mit.edu/spring23/readings/style#_magic_numbers
#
#    priority: 0
#    explanation: ""
#    correctFileRequired: true
#    testResultsFilename: .results.txt
```

```python
#    diffConfiguration:
#       yankedInsertions:
#           - MINIMUM_PASSING_SCORE=65
#       transformers:
#           - regex: \s+
#             replacement: ""
#           - regex: (\(|\))
#             replacement: ""
#       triggeredHintsForUnchanged:
#           - regex: score>=65
#             hint: You need to create a constant instead of comparing
↪  with the magic number, 65.
#             appliedBeforeTransformers: false
#       triggeredHintsForInsertions:
#           - regex: "[A-Z]+(_[A-Z]+)*=65"
#             hint: Constant created for number 65!
#             appliedBeforeTransformers: false
#</yaml>

def passes_course(score):
    '''
    Given grade score, determines if student passes the course.
    >>> passes_course(100)
    True
    >>> passes_course(30)
    False
    '''
    return score >= 65

# DO NOT MODIFY BELOW
if __name__ == '__main__':
```

```python
import doctest
import os
with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
↪  'w') as results:
    results.write(str(doctest.testmod(verbose=True).failed))
```

The metadata is contained within a YAML comment along with the Python source code that is shown to the student. The YAML comment is extracted from the file in the Web Application and the metadata is converted into TypeScript models that help with correctness checking. The student only sees the source code when doing the exercise; the YAML comment is only to be seen by the exercise author.

The YAML metadata consists of the following keys:

- **exerciseId**: Exercise identifier
- **conceptMapIds**: Concept Map containing the exercise
- **conceptIds**: Concept Ids upon which the exercise belongs
- **prompts**: Prompts or hints for the Student to complete the exercise
- **priority**: Index where the exercise should be displayed with respect to the exercises in the concept group
- **explanation**: Explanation of the concept of exercise and the approach used to get to the answer.
- **correctFileRequired**: Boolean denoting whether the exercise requires a secondary source file (also known as the staff file) that has the correct answer. The staff file is compared to the student's solution to determine if the student correctly answer the exercise.
- **testResultsFilename**: Name of file that will store test case results. This is used to check if the student ran test cases and passed them.

- **diffConfiguration**: Contains information that is used to attempt to transform student answer into staff answer. The exercise author can also include Triggered Hints in order to help students if they have a common misconception.

For every exercise, including refactoring exercises, the exercise author writes a prompt in the **prompts** key to give students instructions. The last three keys are important for refactoring exercises. The **correctFileRequired** key is used to tell the tutor whether a correct version of the code is required. The **testResultsFilename** key lets the tutor know if test cases need to be run and where to save the test cases results; if the key is not specified, then the plugin does not require the student to run any test cases, but if there are, then the plugin checks if the student ran the tests and passed them before allowing the student to submit their solution to the Web App. The **diffConfiguration** key stores configuration data for diffs. The exercise author is responsible for curating the diff configuration for each exercise.

6.4.1 diffConfiguration

The **diffConfigration** key maps to a DiffConfiguration typescript model. The model has the following fields:

- **yankedInsertions**: Array of strings that should be pulled out of an insertion before matching occurs.
- **transformers**: Array of transformers
- **triggeredHintsForUnchanged**: Array of triggered hints that will be applied exclusively to unchanged parts of code
- **triggeredHintsForInsertions**: Array of triggered hints that will be applied exclusively to insertions
- **triggeredHintsForDeletions**: Array of triggered hints that will be applied exclusively to deletions

The following sections will explain the purpose of the above fields.

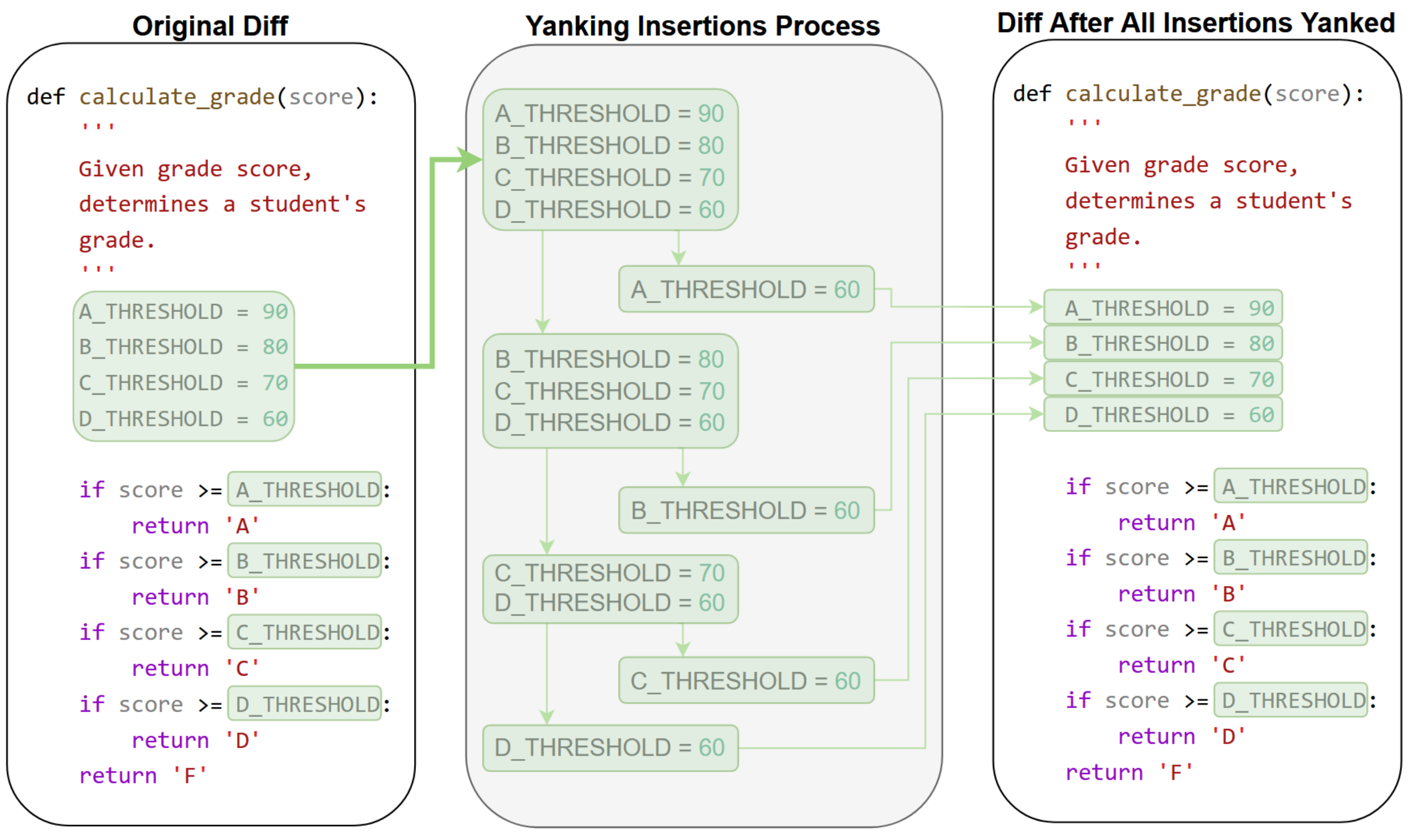
6.4.2 Yanked Insertions

The diff checking and matching procedure is capable of matching insertions and checking if diffs are identical, however, there are cases where the diff generation algorithm groups multiple insertions together if they are near each other. In Figure 6-3, the threshold constants seem to be separate, but without further processing the raw diffs, the constants will be grouped together. This is due to the nature of diff-ing. The diff-match-patch library used to obtain diffs, for example, analyzes changes to two pieces of code and determines differences. Those differences can be insertions or deletions. Insertions and deletions are grouped together based on proximity, however. Unchanged lines of code are also grouped together. This is simply how diff-ing is meant to function. For purposes of the refactoring tutor however, single line insertions and deletions are usually desired. Therefore, before the diff checking and matching procedures are executed, the student and staff diffs are pre-processed in order to extract the desired insertions.

It is not enough to split up the the insertions by newlines. This would work for some cases, but if the exercise author wants to specify that an insertion with newlines should be separate from a group of insertions, then splitting the insertion group by newlines would not work. Also, since the refactoring tutor is designed to be language agnostic, support for languages other than Python is necessary.

To robustly split up a group of insertions, a set of desired insertions must be specified by the exercise author. That is, the exercise author must determine which pieces of code should be pulled out from a group of insertions. The exercise author can specify the insertions in the **yankedInsertions** (since insertions are pulled out or "yanked" from a group of insertions) field in the exercise YAML comment. If the **yankedInsertions** field is not defined, then the diff checking and matching runs as usual. However, if the field is defined, then the Web App uses it to yank insertions from the student and staff diffs. Then, the diff checking and matching occurs. Figure 6-4 shows the process of yanking insertions from a group of insertions in a diff. In the example in Figure 6-4, the yanked insertions that the exercise author would have specified to extract the threshold constants are: `A_THRESHOLD = 90`, `B_THRESHOLD = 90`, `C_THRESHOLD = 90`, and `D_THRESHOLD = 90`.

In summary, yanking insertions from diffs is a necessary pre-processing step before diff matching. It helps make exercises robust to unexpected, but valid inputs from the student.



```
def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    A_THRESHOLD = 90
    B_THRESHOLD = 80
    C_THRESHOLD = 70
    D_THRESHOLD = 60

    if score >= A_THRESHOLD:
        return 'A'
    if score >= B_THRESHOLD:
        return 'B'
    if score >= C_THRESHOLD:
        return 'C'
    if score >= D_THRESHOLD:
        return 'D'
    return 'F'
```

```
def calculate_grade(score):
    '''
    Given grade score,
    determines a student's
    grade.
    '''
    A_THRESHOLD = 90
    B_THRESHOLD = 80
    C_THRESHOLD = 70
    D_THRESHOLD = 60

    if score >= A_THRESHOLD:
        return 'A'
    if score >= B_THRESHOLD:
        return 'B'
    if score >= C_THRESHOLD:
        return 'C'
    if score >= D_THRESHOLD:
        return 'D'
    return 'F'
```

Figure 6-4: Yanking Insertions in Magic Number Exercise

In this example, assuming the exercise author has specified to yank the threshold constant assignments. The original diff shows that the inserted threshold constants are grouped together. The other insertions in the diff aren't grouped together because they are separated by unchanged text. The diff along with the author's specified yanked insertions are used to separate the insertions within the group. This is shown by the yanking insertion process, where author specified insertions are yanked one at a time. The resulting diff has separated insertions, which can then be fed into the diff matching algorithm.

6.4.3 Transformers

Transformers are used in the refactoring tutor to help transform the student solution into a version similar to that of the staff solution. A transformer is represented as a model with two fields: **regex** and **replacement**. The **regex** field stores a regular expression that will attempt to be matched in a given text. If a match is found, then the expression that was matched is replaced by the specified **replacement**. The reason this is needed is because exercises which have many valid answers should allow for student solutions that are different than the staff solutions. The refactoring tutor allows alternate student solutions by using transformers. Specifically, the tutor attempts to apply transformations to the student solution before it can check for correctness. Given the transformer functionality, transformers can help convert the student solution into the staff solution. Transformers were used in the renaming variables exercises. Recall that renaming poor variable names is a subjective process. There are many variable names that can serve as a replacement, some better than others, but it is difficult to draw the line between what is a good or bad variable name. Therefore, the solution is to allow the exercise author to specify transformers that could change potential student solutions to the canonical staff solution. Consider an exercise where the student has to rename a single character variable `t` to `total`. In this case the staff solution would be to change instances of the variable `t` to `total`. However, if the student chooses to use a different variable name, such as `total_sum`, then the exercise author could create a transformer that converts `total_sum` into `total`. The exercise author can create multiple transformers that can convert student solutions to the canonical staff solution. An example of such an exercise is shown below. In the function `avg_of_list()`, the variable `t`, should be renamed. In the exercise metadata, there is a transformer in lines 30-31 that converts variable names that have to do with the concept of total to the variable `total`.

```
1  #<yaml>
2  #   - id: Single-character-variable-name
```

```
#     conceptIds:
#       - python::variable-renaming
#     prompts:
#       - >-
#           Choosing variable names can be tricky sometimes, but you
#           shouldn't resort to variable names that other people (or
#           future you) will find hard to make sense of in the context
#           of your code. As a simple example, it is usually a bad idea
#           to have single character variable names unless it is being
#           used for an index perhaps. Identify and replace the single
#           character variable in the following code with a better, more
#           descriptive name.
#
#           If your first guess doesn't seem to work, try out other
#           variable names you think make sense in the context of the
#           code.
#       - >-
#
    ↪  https://py.mit.edu/spring23/readings/style#_concise_descriptive_names
#
#     priority: 0
#     explanation: ""
#     correctFileRequired: true
#     testResultsFilename: .results.txt
#     diffConfiguration:
#        transformers:
#            - regex: \s+
#              replacement: ""
#            - regex: (\(|\))
#              replacement: ""
#            - regex: (num_sum|num_total|total(_(\w)+)*|((\w)+_)*total)
```

```python
#               replacement: "total"
#</yaml>

def avg_of_list(nums):
    '''
    Given a list of integers, calculates the average.
    >>> avg_of_list([1])
    1.0
    >>> avg_of_list([1, 2, 3, 4, 5])
    3.0
    >>> avg_of_list([3, 6, 9, 16])
    8.5
    '''
    # keeps track of sum
    t = 0
    for num in nums:
        t += num
    return t / len(nums)

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
    with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
     ↪  'w') as results:
        results.write(str(doctest.testmod(verbose=True).failed))
```

The implementation for transformers was relatively simple. The tutor takes the transformers the exercise author made and applies them to both the student and staff solutions. The text in the student and staff solutions is changed to reflect replacements specified by transformers. Currently, all transformers are applied until the code stops changing. This is useful for transformers as it makes them robust to their application order. In other words, if one transformer is trying to remove a whitespace character, and another transformer introduces a space, then the whitespace transformer will delete the space. The tutor will attempt to transform the student solution into the staff solution before checking and matching the diffs.

6.4.4 Triggered Hints for Diffs

Yanked insertions and transformers help make the refactor tutor resilient to unexpected student solutions. However, when the student is unable to get a correct answer, they need guidance or hints. One way that the refactor tutor provides hints is via triggered hints. Triggered hints are hints that are triggered by certain patterns in code. Specifically, the triggers are regexes, just like in transformers. A triggered hint is triggered whenever a provided regex is matched. Just like transformers, triggered hints are created by the exercise author. The following fields are required to define a **TriggeredHint**:

- **regex**: Regex to used to trigger hint
- **hint**: Hint that should be shown to students if regex is triggered
- **appliedBeforeTransformers**: Boolean to denote whether triggered hint should be applied before or after transformers. A true value will apply triggered hints before transformers (this is the default behaviour)

Triggered hints are categorized into three groups: **triggeredHintsForUnchanged**, **triggeredHintsForInsertions**, **triggeredHintsForDeletions** (these are fields in the **diffConfiguration** model, which the exercise author can define). To get hints, each group of triggered hints is applied to the respective diff group (unchanged, insertions, or deletions). **triggeredHintsForUnchanged** are only applied to sections of text that were not changed. **triggeredHintsForInsertions** are only applied to insertions in the text. **triggeredHintsForDeletions** are only applied to deletions in the text. The reason triggered hints were grouped as such is to allow the exercise author to fine tune hints for exercise. It is possible that the exercise author wants a triggered hint just insertions. Sometimes a hint should only be triggered based upon deleted text, not inserted or unchanged text. Having this sort of flexibility helped make exercises more helpful.

Apart from applying triggered hints to either unchanged, inserted, or deleted text, triggered hints could also be applied before or after transformers are applied. The exercise author simply has to specify `true` or `false` in the triggered hints **appliedBeforeTransformers** field. Allowing exercise authors to chose whether to apply triggered hints before or after transformers are applied is useful since transformers can potentially distort the original text. For example, consider the following fields of a whitespace transformer:

- **regex**: \s+
- **replacement**: `""`

The whitespace regex (\s+) matches a sequence of one or more whitespace characters. The whitespaces are then replaced with the empty string. Applying this transformer until it can no longer be applied deletes all whitespace characters from a given text. This is necessary since adding or removing whitespaces should not affect the correctness of the student solution to an exercise in most cases. However, the text after removing the whitespaces will be difficult to reason about for the exercise author. Therefore, it can be useful for the exercise author to specify whether triggered hints should be applied before or after transformers.

6.5 Test Cases

Running test cases is an integral facet to the refactoring tutor. As previously mentioned, the student is required to run test cases (if available) to ensure correct functionality. Each exercise may have a set of test cases to be run before the student can submit their solution. This is important for two main reasons. First, the student should develop an instinct to run test cases whenever a small change has been made to code regardless of whether it was a refactoring change. Making sure tests cases pass when a small change has been made helps prevent bugs. Secondly, ensuring that the student passes the test cases helps prune many invalid submissions that the checker determines as correct. If the test cases pass, the checker will assume that the student code is functional. The diff checking, diff matching, and tests cases all work together to make the refactoring tutor robust.

Chapter 7

Deployment

The refactoring tutor was deployed to two different groups of programmers over the course of a couple of months. The first group consisted of novice programmers who are currently enrolled in 6.1010: Fundamentals of Programming at MIT. This course is taken by most computer science majors since it is a required course in the curriculum at MIT. Many of the students in the course do not have much experience programming, and possibly even less experience with refactoring. Students in 6.1010 could benefit quite a bit from learning to refactor their code, since writing code that is difficult to understand or prone to bugs will result confusion and even when asking for help, staff may find it difficult to debug when code is messy. The refactoring tutor was also deployed to the staff of 6.1010, which consisted of teaching and lab assistants. The TAs and LAs consist of intermediate to advanced programmers. Deploying the tutor to the TAs and LAs was useful for gathering feedback about the tutor and the exercises and concepts about refactoring that it attempts to teach students.

7.1 First Deployment

The first round of deployment of the refactoring tutor was in early March 2023. At this time, the refactoring tutor supported only Level 0 and Level 1 exercises. A total of 14 exercises, providing practice for 10 concepts, were available. Before deploying the tutor, it was necessary to identify which students needed the most help with code style. Fortunately, lab assignments for 6.1010 had a style component. That is, student's code was evaluated for style. The style checker used a combination of Pylint and Radon complexity checks to evaluate the style of the student's code. Pylint is a static code analysis tool for Python; it helps identify poor variable names, boolean laundering, and long lines for example. Radon is another tool supported for Python that determine cyclomatic complexity, a measure of the number of linearly independent paths through a program. In other words, cyclomatic complexity is a measure of how complex a given piece of code is. The complexity is determined via multiple metrics (many conditional statements and for / while loops contribute to complexity, for example).

Roughly 80 students were contacted to participate and try out the refactoring tutor based on code style data that was collected on assignments. The students contacted were flagged for mediocre code style. Unfortunately, there were only two participants. The participants however, were able to completely finish all of the Level 0 and Level 1 exercises in about 30 minutes.

7.2 Second Deployment

The refactoring tutor was deployed online for students in 6.1010 in the beginning of April. Students were given instructions on how to install the tutor and begin working on exercises. Additionally, there was an extra help session every week, where students were encouraged to try out the refactoring tutor if they had completed the session's task. The tutor featured the final Level 2 exercises from the refactoring tutor. A total of ten new exercises and eight new concepts were included in the second version of the tutor. These exercises were more challenging than those included in the first round of deployment. In April, an additional 20 students completed at least one exercise in the latest version of the refactoring tutor. A few students managed to complete all the exercises. Most students did not complete them all, however.

Apart from the students, the refactoring tutor was also deployed to the staff of 6.1010, as mentioned previously. About 50 staff attempted the refactoring tutor. The staff was instructed to complete the tutor as they would if they were also students, and then go back and attempt to find bugs or other unintended behavior found in the tutor. There were some technical difficulties in installing the tutor according to various staff members, but within roughly an hour, about half of the staff was able to install the tutor and complete Level 0, Level 1, and Level 2 exercises. During the session, 45 feedback responses were collected. The feedback will be discussed in the evaluation section.

Chapter 8

Evaluation

This section discusses the refactoring tutor feedback gathered from staff and the results of deploying the refactoring tutor in 6.1010: Fundamentals of Programming. The refactoring tutor was used by most of the course staff and many members claimed the tutor would be a useful tool for students. The staff did provide feedback on certain exercises and some found minor bugs in a few exercises. With the infrastructure of the tutor however, it was possible to modify exercises to take into account the staff feedback and fix the bugs.

The students who used the tutor were able to get past the Level 0 and Level 1 exercises, with some amount of success with the Level 2 exercises. Since there is not enough student data, quantitative conclusions regarding the effectiveness of the refactoring tutor on teaching students about refactoring won't be made. Instead, this section will focus mainly on the qualitative aspects of the refactoring tutor and the staff feedback.

8.1 Student Results

Over the past few months, 21 students used the refactoring tutor and made progress on the exercises. In total, the 21 students collectively submitted their answers 362 times. 201 of the submissions were marked as correct, while 161 of the submissions were marked as wrong. Every student was able to pass the tutorial exercise. 11

students were able to finish at least half of the exercises. The first half of the exercises correspond to Level 0 and Level 1 exercises which aren't very challenging, so it makes sense that a good portion of the student were able to finish those. Three students were able to complete the majority of the exercises. Figure 8-1 shows how many exercises were completed by how many students.

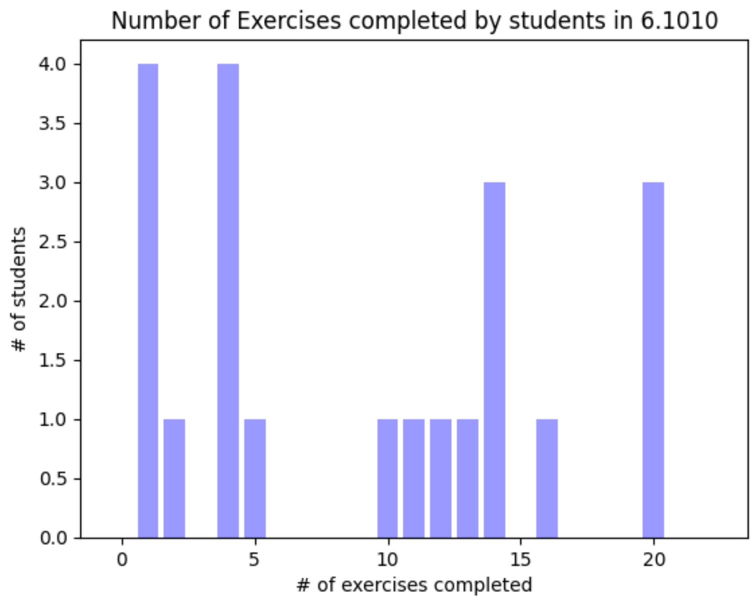


Figure 8-1: Exercises Completed by students in 6.1010

There were a few exercises that were seemingly confusing to the students. Figure 8-2 shows the number of passed and failed attempts per exercise. Based on Figure 8-2, there were six exercises that had more failed attempts than successful attempts. These exercises include Boolean-laundering-if-statements, Long-line-sobel-filter, Single-character-variable-name, Single-character-variables-compound-interest, Register-user-never-nest, and Tuple-unpacking-in-loops. The next sub-sections will discuss the types of issues students encountered in the mentioned exercises, along with fixes to make the exercise better. The Long-line-sobel-filter was not attempted nearly as much as the other exercises, therefore, it will be excluded from the discussion.

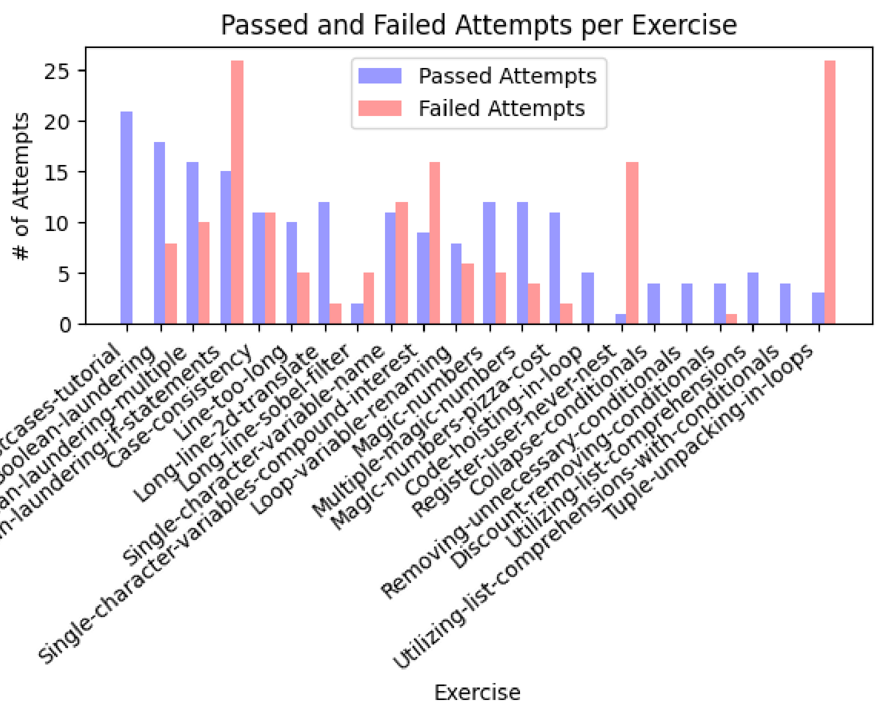


Figure 8-2: Passed and Failed Attempts per Exercise from students

8.1.1 Boolean-laundering-if-statements

The Boolean-laundering-if-statement exercise had 15 successful attempts and 26 failed attempts. The exercise is shown below:

```
#<yaml>
#  - id: Boolean-laundering-if-statements
#    conceptIds:
#      - python::boolean-laundering-in-if-statements
#    prompts:
#      - >-
#          Notice in the following exercise that <b>score >= 65</b> is
#          used as the check for a passing grade. Then a boolean value
#          is returned, but the if-condition is itself a boolean value
#          after being evaluated. Think of what you can change to make
#         the code more succint and readable.
#      - >-
#
↪  https://py.mit.edu/spring23/readings/style#_boolean_laundering
#
#    priority: 2
#    explanation: ""
#    correctFileRequired: true
#    testResultsFilename: .results.txt
#    diffConfiguration:
#       transformers:
#           - regex: \s+
#             replacement: ""
#           - regex: (\(|\))
#             replacement: ""
#       triggeredHintsForUnchanged:
#           - regex: return True
#             hint: Do you explicitly need to return True?
#             appliedBeforeTransformers: true
#           - regex: return False
#             hint: Do you explicitly need to return False?
#             appliedBeforeTransformers: true
#</yaml>

def passes_course(score):
    '''
    Given grade score, determines if student passes the course.
    >>> passes_course(100)
    True
    >>> passes_course(30)
    False
```

```python
    '''
    if score >= 65:
        return True
    return False

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
    with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
     ↪  'w') as results:
        results.write(str(doctest.testmod(verbose=True).failed))
```

The solution to this exercise requires removing the verbose return statements:

```python
def passes_course(score):
    '''
    Given grade score, determines if student passes the course.
    >>> passes_course(100)
    True
    >>> passes_course(30)
    False
    '''
    return score >= 65

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
    with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
     ↪  'w') as results:
```

```
16          results.write(str(doctest.testmod(verbose=True).failed))
```

It turns out that 19 of the 26 of the incorrectly marked submissions had the correct refactoring that the exercise was looking for. However, the student solutions were somehow modified such that docstrings used double quotes instead of single quotes as is used in the starter code and the solution. For example, one of the student submissions is shown below:

```
1  def passes_course(score):
2      """
3      Given grade score, determines if student passes the course.
4      >>> passes_course(100)
5      True
6      >>> passes_course(30)
7      False
8      """
9      return score >= 65
10 
11 # DO NOT MODIFY BELOW
12 if __name__ == '__main__':
13     import doctest
14     import os
15     with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
       ↪  'w') as results:
16         results.write(str(doctest.testmod(verbose=True).failed))
```

Lines 2 and 8 are different between the staff solution and the student submission. Changing single quotes to double quotes shouldn't affect the correctness of the exercise, since the intended refactoring was done by most of the students, therefore, the student submission should have been marked correct. This could easily be fixed by adding a transformer that converts all double quotes to single quotes in the metadata of the exercise. For example, the new metadata of the exercise could look like the following:

```
#<yaml>
#  - id: Boolean-laundering-if-statements
#    conceptIds:
#      - python::boolean-laundering-in-if-statements
#    prompts:
#      - >-
#         Notice in the following exercise that <b>score >= 65</b> is
#         used as the check for a passing grade. Then a boolean value
#         is returned, but the if-condition is itself a boolean value
#         after being evaluated. Think of what you can change to make
#         the code more succint and readable.
#      - >-
#
↪  https://py.mit.edu/spring23/readings/style#_boolean_laundering
#
#    priority: 2
#    explanation: ""
#    correctFileRequired: true
#    testResultsFilename: .results.txt
#    diffConfiguration:
#       transformers:
#           - regex: \s+
#             replacement: ""
#           - regex: (\(|\))
#             replacement: ""
#           - regex: \"
#             replacement: "\""
#       triggeredHintsForUnchanged:
```

```
#           - regex: return True
#             hint: Do you explicitly need to return True?
#             appliedBeforeTransformers: true
#           - regex: return False
#             hint: Do you explicitly need to return False?
#             appliedBeforeTransformers: true
#</yaml>
```

Single quotes might have been changed to double quotes in the students' submissions unintentionally by a VSCode extension that autoformats code. This is the most likely scenario since the students probably didn't manually change the single quotes to double quotes.

8.1.2 Single-character-variable-name

There were a variety of different answers in the Single-character-variable-name exercise. The exercise file is shown below:

```
#<yaml>
#  - id: Single-character-variable-name
#    conceptIds:
#      - python::variable-renaming
#    prompts:
#      - >-
#         Choosing variable names can be tricky sometimes, but you
#         shouldn't resort to variable names that other people (or
#         future you) will find hard to make sense of in the context
#         of your code. As a simple example, it is usually a bad idea
#         to have single character variable names unless it is being
#         used for an index perhaps. Identify and replace the single
#         character variable in the following code with a better, more
#         descriptive name.
#
#         If your first guess doesn't seem to work, try out other
#         variable names you think make sense in the context of the
#         code.
#      - >-
#
↪  https://py.mit.edu/spring23/readings/style#_concise_descriptive_names
#
#    priority: 0
#    explanation: ""
#    correctFileRequired: true
#    testResultsFilename: .results.txt
#    diffConfiguration:
#       transformers:
#           - regex: \s+
#             replacement: ""
#           - regex: (\(|\))
#             replacement: ""
#           - regex: (num_sum|num_total|total(_(\w)+)*|((\w)+_)*total)
#             replacement: "total"
#</yaml>

def avg_of_list(nums):
    '''
    Given a list of integers, calculates the average.
    >>> avg_of_list([1])
    1.0
    >>> avg_of_list([1, 2, 3, 4, 5])
    3.0
    >>> avg_of_list([3, 6, 9, 16])
```

```python
    8.5
    '''
    # keeps track of sum
    t = 0
    for num in nums:
        t += num
    return t / len(nums)

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
    with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
     ↪ 'w') as results:
        results.write(str(doctest.testmod(verbose=True).failed))
```

As has been seen before, the idea of this exercise is to rename the variable `t` into `total`. Some students got the exercise wrong because they renamed `t` to `sum`. This is intentional, since renaming a built-in function (sum is a built-in function in Python) is not good practice. Another student used the variable name `running`, which isn't included in the transformer regex that transforms variable names to `total`. It's quite subjective, however, `running` isn't as expressive as `total` or `total_sum`. However, one student did use `running_sum`, which is a better variable than `t` or `running`, however the tutor marked it wrong. To allow for such a variable name, the transformer that converts variable names to `total` could be modified to include `running_total`. The new diff configuration in the exercise metadata could look something like the following:

```python
# diffConfiguration:
#    transformers:
#        - regex: \s+
#          replacement: ""
#         - regex: (\(|\))
#           replacement: ""
#         - regex:
  ↪  (num_sum|num_total|total(_(\w)+)*|((\w)+_)*total|running_sum|((\w)+_)+sum)
#           replacement: "total"
```

Notice that `running_sum` is now included in the transformer regex. A more general regex that is included in the transformer regex however is `((\w)+_)+sum`.

Four out of the 12 failed attempts involved renaming the variable `nums` to `numbers`, which is subjectively a good change. However, the exercise is only expecting the variable `t` to be renamed. To prevent confusion, the prompt could be modified to explicitly mention only to rename single character variable names.

8.1.3 Single-character-variables-compound-interest

The Single-character-variables-compound-interest exercise had a student outlier where 11 failed attempts were made. Given that the mentioned exercise is a variable renaming exercise and therefore somewhat open-ended, it was somewhat expected that student would struggle with renaming variable names. In fact, the outlier student submitted feedback for the exercise and mentioned that they tried multiple variable names that weren't working. The exercise the student was struggling with is shown below:

```
#<yaml>
#  - id: Single-character-variables-compound-interest
#    conceptIds:
#      - python::variable-renaming
#    prompts:
#      - >-
#          In this example, there are multiple single character variable
#          names which might be confusing those who aren't familiar with
#         compounding interest. Identify and replace the single
#         character variables in the following code with better, more
#         descriptive names.
#
#    priority: 1
#    explanation: ""
#    correctFileRequired: true
#    testResultsFilename: .results.txt
#    diffConfiguration:
#       transformers:
#           - regex: \s+
#             replacement: ""
#           - regex: (\(|\))
#             replacement: ""
#           - regex:
↪   (starting(_(\w)+)*|start(_(\w)+)*|((\w)+_)*start|((\w)+_)*starting)
#             replacement: "starting"
#           - regex: (principle(_(\w)+)*|((\w)+_)*principle)
#             replacement: "starting"
#           - regex:
↪   (interestRate|rateInterest|rate(_(\w)+)*|((\w)+_)*rate)
#             replacement: "rate"
#           - regex: (time(_(\w)+)*|((\w)+_)*time)
#             replacement: "time"
#</yaml>

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

```python
    >>> compound_interest_monthly(100, 0.10, 12)
    10.471306744129677
    >>> compound_interest_monthly(2000, 0.05, 24)
    209.8826711166539
    '''
    amount = s * ( pow( (1 + r / MONTHS), t ) )
    return amount - s

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
    with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
     ↪  'w') as results:
        results.write(str(doctest.testmod(verbose=True).failed))
```

I extracted student logs for the variable renaming exercise to determine why the student's submissions were being marked wrong. As expected, it turns out that the student used a variable name that the exercise was not expecting. Specifically, the student was replacing `t` with `elapsed_months`, which was not accepted by the tutor. The variable name `elapsed_months` is completely valid. Updating the exercise to allow for the `elapsed_months` variable name required adding a transformer that converted `elapsed_months` to `time`. The infrastructure was built to easily add transformers to detect and change valid variable names into those of the staff solution.

8.1.4 Register-user-never-nest

Another confusing exercise for the students was the Register-user-never-nest exercise, one of the hardest exercises. Only one student was able to get it right. Not many students understood what they had to do, since this concept is rather niche for students. The exercise is shown below:

```
#<yaml>
#  - id: Register-user-never-nest
#    conceptIds:
#      - python::reducing-nesting
#    prompts:
#      - >-
#          In this exercise, you can see that there is a bit of nesting
#          in the logic to register a user. This is a simple example,
#          but in complex functions that have a lot of conditional
#          logic, nesting can get terrible. One way to reduce that
#          however, is to do all the error checking first. This usually
#          requires inverting some conditionals that deal with returning
#          early. In this example, we return None whenever we hit an
#          unhappy case. After returning, it is then unnecessary to nest
#          code anymore, which helps with clarity of the code. This
#          process splits up the code into the initial error checking
#          part, and then the actual logic that the function should
#          generally execute.<br><br>
#    priority: 1
#    explanation: ""
#    correctFileRequired: true
#    testResultsFilename: .results.txt
#    diffConfiguration:
#       yankedInsertions:
#           - "iflenparts!=2:returnNone"
#           - "ifuser_id<=0:returnNone"
#           - "ifnotlenparts==2:returnNone"
#           - "ifnotuser_id>=0:returnNone"
#       transformers:
#           - regex: \s+
#             replacement: ""
#           - regex: (\(|\))
#             replacement: ""
#       triggeredHintsForUnchanged:
#           - regex: "iflenparts==2:(.)*ifuser_id>=0:"
#             hint: The length 2 parts condition and non-negative ids
↪  condition are still nested.
#                   The goal is un-nest the conditions.
#</yaml>
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.
    >>> validate_and_register('1, ben')
    (1, 'ben')
    >>> validate_and_register('100, alice')
    (100, 'alice')
    """
    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
```

```python
    else:
        return None

# DO NOT MODIFY BELOW
if __name__ == '__main__':
    import doctest
    import os
    with open(f'{os.path.expanduser("~")}/.praxistutor/.results.txt',
    ↪   'w') as results:
        results.write(str(doctest.testmod(verbose=True).failed))
```

This goal behind this exercise is explained in the Excessive Nesting sub-section of the Code Smells section. Student were able to reduce nesting based on their submissions, however, not in the way the staff solution expected. For example, one of the student submissions is shown below:

```python
def validate_and_register(user_info):
    parts = user_info.replace(" ", "").split(",")

    if len(parts) != 2 or int(parts[0]) < 0:
        return None

    user_id = int(parts[0])
    user_name = parts[1]

    return (user_id, user_name)
```

This does manage to reduce nesting. There is only one level of nesting here. However, it is marked incorrect because the two error conditionals are combined into one. In this case it's not a good idea to combine the two conditions, since the `int(parts[0])` is repeated twice. The intent of this exercise was that students have two different if-statements, one for checking the length of `parts` and the other for checking the numerical values of the user id.

Another student submission, which was very close to being marked correct is shown below:

```python
def validate_and_register(user_info):
    parts = user_info.replace(" ", "").split(",")

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

In this case, the student has the first error checking if-statement. The student has the right idea to return a user id and user name if the `user_id` >= 0, but to follow the paradigm explained in the exercise, the student was supposed to return fast and check if `user_id < 0`.

To help fix this issue, more clear instructions were added to the exercise description. Specifically, the prompt now states that there must be two error checking conditions that must return `None`. New hints are also included. If a student presses the "get hint" button, an example of returning fast with the first error condition is shown with the hopes that they are able to figure out how to refactor the rest of the function. It was difficult to add triggered hints for this exercise since student solutions tend to deviate a lot from the staff solution, therefore better instructions and manual hints were used to help students with this exercise.

8.1.5 Tuple-unpacking-in-loops

The Tuple-unpacking-in-loops exercise was also confusing for students. Only three students were able to correctly answer the exercise. This may partially be due to the fact that not many students got to the last exercise. One of the students that was not able to pass the exercise submitted feedback. The student attempted the exercise over 20 times without success. Most of the student submissions are the same (the student was not sure why they were getting the exercise wrong and perhaps submitted the same code over and over). However, the reason the student was marked incorrect was because they were using a list comprehension as their solution. The tuple unpacking exercise that the student was unable to get right is shown below:

```python
# EXERCISE STARTER CODE
def calculate_z_scores(data):
    '''
    Calculates z-scores for each entry in data. Each entry
    in data is a tuple consisting of (mean, x, std deviation).
    '''
    z_scores = []
    for entry in data:
        # z-score is calculated by (x - mean) / std deviation
        z_scores.append((entry[1] - entry[0]) / entry[2])
    return z_scores

# EXERCISE STAFF SOLUTION
def calculate_z_scores(data):
    '''
    Calculates z-scores for each entry in data. Each entry
    in data is a tuple consisting of (mean, x, std deviation).
    '''
    z_scores = []
    for mean, x, std_dev in data:
        # z-score is calculated by (x - mean) / std deviation
        z_scores.append((x - mean) / std_dev)
    return z_scores

```

```python
# SUGGESTED STUDENT SOLUTION
def calculate_z_scores(data):
    '''
    Calculates z-scores for each entry in data. Each entry
    in data is a tuple consisting of (mean, x, std deviation).
    '''
    return [(x - mean) / std_dev for mean, x, std_dev in data]
```

The intended solution was to simply replace the `entry` variable with `mean`, `x`, `std_dev` . The student made an astute observation and realized the code could be refactored into a list comprehension. The student code is more concise than the staff solution, which arguably makes the student code better than the staff code. Another reason the student might have defaulted to list comprehensions is because the exercises preceding the tuple unpacking exercise focus on list comprehensions. The exercise group dealing with idiomatic python code contained both list comprehension and tuple unpacking concepts. So it's likely that, the student was misguided by the ordering of the exercises. To help prevent scenarios like this, two main changes were made to the exercise. Firstly, the tuple unpacking exercise now precedes the list comprehension exercises. Secondly, the exercise now mentions to focus on only unpacking tuples and not list comprehensions, since the goal of the tutor is focus in on certain refactoring concepts.

8.2 Staff Results

Compared to the student results, it seems like staff were more successful. That is expected since staff have more experience with coding and refactoring. Figure 8-3 shows the number of exercises the staff completed. 14 staff members completed all the exercises. 27 staff completed 15 or more exercises. Some staff couldn't make it through the entire tutor due to installation issues and bugs associated with having reformatting extensions in VSCode. However, it seems that the refactoring exercises were doable by the staff. The staff was also instructed to attempt to break the tutor and find any bugs after they finished the set of exercises.

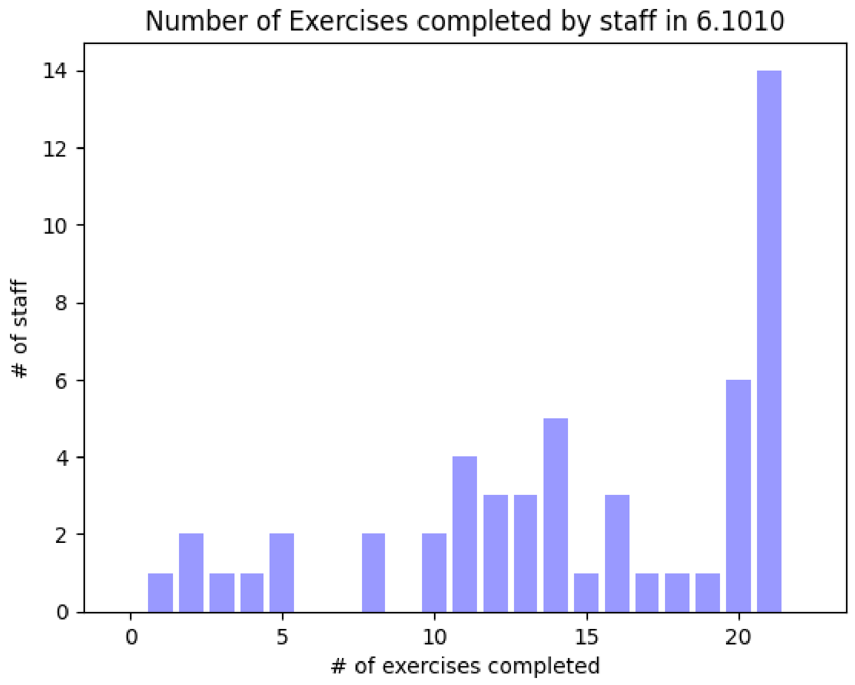


Figure 8-3: Number of Exercises completed by staff

The staff also seemed to have struggled with the same exercises the students struggled with. As can be seen in Figure 8-4, the same three exercises that the students struggled with were the exercises with the most failed attempts by the staff. The exercises were: Single-character-variables-compound-interest, Register-user-never-nest, and Tuple-unpacking-in-loops. The Register-user-never-nest exercise was failed the most by the staff as it was by the students. More explanations were added to the mentioned exercise in hopes of making it more clear to students and staff who use the tutor.

8.3 Staff Feedback

The staff provided helpful feedback after they tried out the refactoring tutor. There were 45 individual responses collected. Out of the 45 responses, two were about hints,

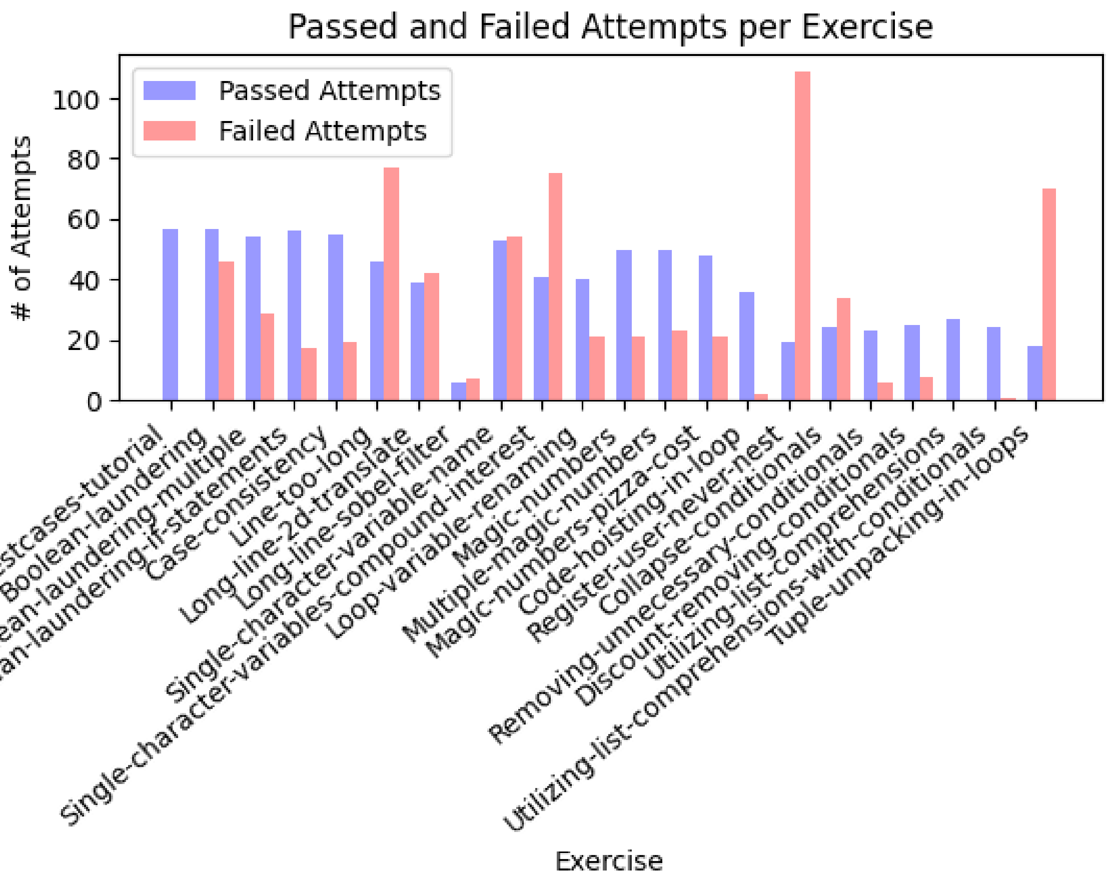


Figure 8-4: Passed and Failed Attempts per Exercise from staff

six were about the problem descriptions, and 13 were about potential bugs. The other responses were positive feedback about the tutor and how it could be used to help students in 6.1010. Most of the feedback received mentioned how either the problem description or hints could be made clearer.

8.3.1 Hints

The feedback for hints in the refactoring tutor mentioned that more explicit hints should be given. One user wrote, "hints should explain what exact changes to the program should be made." In some cases, this was true. The hints for some exercises were sparse (some hints linked to readings, instead of explicitly mentioning what exact changes should be made), so understandably a student who is struggling with the exercises would prefer explicit hints. However, it is difficult to balance how many and what hints to give, since the idea for the tutor is to get the student thinking about how to refactor via exercises. The goal is not to tell the student how to solve each exercise. Hints for the exercises were revised, however. Hints were added to the most difficult exercises suggested by the student and staff results. Given the tutor infrastructure, adding hints to exercises is trivial. The exercise author simply has to create either triggered hints or explicit hints in the exercise definition.

8.3.2 Exercise Description

Similar to hints, the staff made it clear that more explicit instructions on the exercises would be better. Some staff suggested rewording some instructions and descriptions. For example, in the long line exercises, it isn't clear that brackets, parentheses, or curly braces could be separated by newlines in Python. The student probably wouldn't know this, making the exercise confusing. This feature of Python was originally mentioned as a hint, but now is part of the exercise descriptions of the long line exercises. Another staff member claimed that it was confusing not knowing what to rename variables in the renaming variables exercises. They claimed that explicit instructions on what to rename variables would have been great. Although it might be great for completing the exercises, it might not be the best for learning. Having explicit variable names in the instructions for such exercises defeats the purpose of the exercise. However, for renaming variable exercises, perhaps there should be a word bank with commonly chosen words that aren't the best variable name choice. This is difficult since naming variables is partially subjective. A variable name that seems bad to some people will seem good to others.

The exercises descriptions were revised to include more details about what is required and explanations about subtle Python features. This was done trivially, since changing the exercise descriptions requires only changing the prompt in the exercise YAML comment.

8.3.3 Exercise Bugs

Exercise bugs were bugs found by the staff that involved getting an exercise wrong, even though the submitted code was refactored correctly, but in a way that deviated from the exercise's staff solution. An example of this is demonstrated in the tuple unpacking exercise shown below.

```python
 1  # EXERCISE STAFF SOLUTION
 2  def calculate_z_scores(data):
 3      '''
 4      Calculates z-scores for each entry in data. Each entry
 5      in data is a tuple consisting of (mean, x, std deviation).
 6      '''
 7      z_scores = []
 8      for mean, x, std_dev in data:
 9          # z-score is calculated by (x - mean) / std deviation
10          z_scores.append((x - mean) / std_dev)
11      return z_scores
12
13  # SUGGESTED SOLUTION
14  def calculate_z_scores(data):
15      '''
16      Calculates z-scores for each entry in data. Each entry
17      in data is a tuple consisting of (mean, x, std deviation).
18      '''
19      z_scores = []
20      for (mean, x, std_dev) in data:
21          # z-score is calculated by (x - mean) / std deviation
22          z_scores.append((x - mean) / std_dev)
23      return z_scores
```

Notice the difference between lines 8 and 20. The suggested solution was marked incorrect since it included parentheses and deviated from the exercise staff solution. However, it shouldn't be marked wrong, since the refactoring concept of tuple unpacking was used successfully in the exercise. It required adding a transformer that would remove parentheses from text. It might seem weird to remove all parentheses, since the semantic meaning is different depending on the context. In the example above, the parentheses don't matter. The code will work without them. When parentheses are used to call a function, for example, they matter quite a bit. Removing those parentheses would raise an exception. The exercise checkers however don't need to worry about correctness of the code however, since that is handled by the test cases. All submitted code is code that passed the test cases. Therefore, in order to fix the bug, adding a parenthesis transformer works.

Also it's worth mentioning that some staff suggested that the tuple unpacking exercise above can be done using a list comprehension, but as mentioned in the Student Results section, the goal for the tutor is to focus on a particular refactoring technique at a time, therefore the staff solution was unchanged. However, the exercise description was modified to include instructions to focus on tuple unpacking, not creating list comprehensions. Although, it is a neat way to refactor the code. Perhaps future versions of the refactoring tutor could include multiple refactoring techniques.

8.4 Infrastructure Bugs

Many staff had issues with the exercises because they had one or more VSCode extensions that would automatically reformat their code on save. This was especially noticeable for the long line exercises. Staff reported code being changed on save even though the staff didn't want to make such changes. This is of course a feature of the VSCode extensions that auto-format code, however, it is problematic for the refactoring exercises, especially because the exercises rely on the exact submitted text. It was found that staff who were having issue with the exercises submitted answers that were nearly correct according the exercise checkers. The issue was the difference between single quotes and double quotes. It is assumed that the VSCode extensions were the culprit of modifying the exercises to have single quotes instead of double quotes that were contained in the starting exercise code. The staff likely wouldn't have changed all double quotes to single quotes intentionally, especially since none of the exercises instructed them to do so. This was a subtle bug, but it can potentially be fixed with a double quote transformer. Although, that is perhaps not the best solution to this issue. The main problem affecting the correctness of exercises is the VSCode extensions. To make the refactoring tutor more robust, extensions need to be deactivated when doing the exercises, either automatically or manually. This will prevent any future bugs having to do with auto-formatting.

Chapter 9

Discussion

The tutor shows promise in teaching students how to refactor based on the completion of exercises by some students and staff in 6.1010. Exercises have been shown to be rather robust, and when there are bugs with exercises, modifying them is not hard. Based on the the evaluation of the tutor, most of the bugs, or issues found with the exercises were or could be solved by modifying the exercise prompt, hints, or transformers. There was a lot of staff feedback on the exercise prompts for example. Staff mentioned that there were exercises where the instructions were unclear. Changing the exercise prompts only requires changing the prompt in the exercise's YAML comment. This applies to hints also. Adding manual hints can be done by adding to the exercise's YAML comment. Triggered hints could be modified by changing the diff configuration field in the YAML comment. Some students had trouble with the renaming variable exercises as was expected. Some used unexpected, but good variable names, therefore adjusting the exercises to include more acceptable variable names was needed. This was also not hard, and just required adding more transformers to the diff configuration of the exercises.

The data gathered from the students suggested that the exercises were doable, although more data is needed to determine if the exercises are confusing or not. Not all students finished the exercises, however the tutor was optional for students in 6.1010. Perhaps with some incentives in a future semester of 6.1010, will students be determined to finish the exercises. The staff were more successful in completing the exercise. A one hour time slot was allotted during a staff training day to have the staff go through the exercises. It was expected that the staff would have more success than the students, since they have more experience with coding.

9.1 Future Improvements

There are still many aspects to improve on, though. To prevent bugs stemming from unintended auto-formatting, VSCode extensions should be deactivated; or at least the extensions that auto-format code. This probably has to be handled in the plugin. It may be possible to automatically turn the extensions off when using the refactoring tutor, however if it is not possible, then another solution is to check if there are any formatting extensions and let the user know. The tutor could display a warning message that prompts the user to turn off the extensions.

Another potential upgrade for the tutor is to include "compilation" errors. That is, whenever an exercise is marked incorrect, the tutor will pin-point exactly where the issue is by underlining a line(s) of code for example. Having this will make exercises more intuitive and will help student understand what is wrong with their code. Implementing this may require further analyzing the student diff and determining what parts of the diff were missing or incorrectly included.

An addition to the tutor that would also be helpful to the students, would be a streamlined process where students could submit answer suggestions for exercises. In the variable renaming exercises, there was a lot of student frustration that their chosen variable names didn't work. Some of the student's chosen variable names were valid however, therefore, instead of just marking the exercise as wrong, having a way for the student to submit suggestions for valid variable names would be useful. The process could involve various staff members where they approve the suggestion or provide feedback on why the student's suggestion was rejected.

To help exercise authors develop and modify exercises a UI that assists in creating transformers or triggered hints could be made. Editing exercises via the YAML file works, however it would be more user-friendly if the process to do so included some graphical interface.

Chapter 10

Conclusion

In this thesis, a novel tool to help students learn about refactoring and improve their coding style was developed. The refactoring tutor was a VSCode extension that presented student with various types of refactoring exercises. Concepts spanning from removing boolean laundering to writing more idiomatic Python code were included in the tutor. Since refactoring and style is partially a subjective concept, the exercises has to focus on very specific refactoring concepts. Each exercise provided practice with a single concept.

In order to develop refactoring exercises, additions to the existing Praxis Tutor infrastructure were made. The most important addition was the creation of the diff checking and matching algorithms that compared student diffs and staff diffs. The algorithm was used by most exercises. It was robust for the most part, however, the algorithm did use additional configurations set by the exercise author to make the tutor more resilient to edge cases and bugs. A configuration structure was included to allow the exercise author to fine tune the exercise.

Although the results in this paper did not show conclusive evidence that the tutor helped students learn refactoring, it still has promise. Not enough students participated and tried out the tutor, however, if the tutor was assigned for example, in an introductory programming class, then more student data could be gathered and possible show that the tutor does help students learn refactoring. The staff for 6.1010: Fundamentals of Programming play-tested the tutor and provided feedback.

The feedback was used to improve the tutor, and most the feedback involved making a few modifications to the exercises with the infrastructure that was developed for the refactoring tutor. Most of the staff was able to complete most of the refactoring exercises. With a bit more testing and tuning, the tutor has potential to help students practice and learn important refactoring skills.

Bibliography

[1] Patrick Beer. *Code Context based Generation of Refactoring Guidance.* PhD thesis, 02 2019.

[2] Marcus Ciolkowski, Valentina Lenarduzzi, and Antonio Martini. 10 years of technical debt research and practice: Past, present, and future. *IEEE Software*, 38(6):24–29, 2021.

[3] CodeAesthetic. *Why You Shouldn't Nest Your Code.* YouTube, Dec 2022.

[4] Anders Ericsson. *Peak : secrets from the new science of expertise / Anders Ericsson and Robert Pool.* Houghton Mifflin Harcourt, Boston, 2016 - 2016.

[5] Cruz Izu and Shrey Chandra. Are we there yet? novices' code smells linked to loop constructs. pages 1151–1151, 03 2022.

[6] Cruz Izu, Paul Denny, and Sayoni Roy. A resource to support novices refactoring conditional statements. 07 2022.

[7] Amandeep Kaur and Manpreet Kaur. Analysis of code refactoring impact on software quality. *MATEC Web of Conferences*, 57:02012, 01 2016.

[8] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 562–568, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling (SoSyM)*, 6, 08 2007.

[10] Mohammed Misbhauddin and Mohammad Alshayeb. Uml model refactoring: A systematic literature review. *Empirical Software Engineering*, 20, 02 2013.

[11] Suzanne Smith, Sara Stoecklin, and Catharina Serino. An innovative approach to teaching refactoring. volume 38, pages 349–353, 03 2006.

[12] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015.

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]