



Usage of Generative AI Based Plugin in Unit Testing

Evaluating the Trustworthiness of Generated Test Cases
by Codiumate, an IDE Plugin Powered by GPT-3.5 & 4

Ali Reza Nazari
Bow Nannicha Thunell

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Software Engineering. The thesis is equivalent to 10 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Authors:

Ali Reza Nazari

E-mail: alna20@student.bth.se

Bow Nannicha Thunell

E-mail: both21@student.bth.se

University advisor:

Lecturer Nina Dzamashvili Fogelström

Department of Software Engineering

Abstract

Background: Unit testing is essential in software development, ensuring the functionality of individual components like functions and classes. However, manual creation of unit test cases is time-consuming and tedious, impacting testing efficiency and reliability.

Problem: Automated unit test generation tools such as EvoSuite and Randoop have addressed some challenges, but they're limited by language specificity and predefined algorithms. Generative AI tools like ChatGPT and GitHub Copilot powered by OpenAI's GPT-3.5/4 offer alternatives, but face limitations like user input reliance and operational inconveniences.

Solution: CodiumAI's Codiumate IDE plugin aims to mitigate these limitations, making code quality assurance easier for developers. This study evaluates Codiumate's trustworthiness in generating unit tests for the Python functions.

Method: We randomly selected thirty functions from OpenAI's HumanEval dataset, and wrote selection criteria for relevant test cases based on each function's doc string to evaluate Codiumate's trustworthiness using metrics such as Relevance Score, false positive rate, and result consistency rate.

Result: Among all of the suggested test cases by Codiumate, 208 unit tests, which consists of 48% of suggested test cases that were relevant. 70% of assertions from these test cases strictly meet selection criteria, while the other 30% while relevant were selected due to our basis and experience in software testing. The average false positive rate is 15%. Function groups that have higher Relevance Scores are non-mathematical nature, and simple dependencies. High false positives arise in functions with string and float parameters. All generated unit tests are syntax-error-free, with 20% fail and 80% passed in all five test execution.

Conclusion: Codiumate demonstrates potential in automating unit test generation, offering a convenient means to support developers. However, it is not yet fully reliable for critical applications without developer oversight. Continued refinement and exploration of its capabilities are essential for Codiumate to become an indispensable asset in unit test generation, enhancing its trustworthiness and effectiveness in the software development process.

Keywords: Codiumate, ChatGPT, trustworthiness, unit test, test case generation.

Acknowledgments

We would like to extend our gratitude to our supervisor, Nina Dzamashvili Fogelström, for her invaluable guidance, support, and insightful feedback throughout the process of crafting this thesis.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Background	1
1.1.1 Trustworthiness in the Context of this Thesis	2
1.2 Contribution	2
1.3 Scope	2
2 Research Questions	4
3 Related Work	6
4 Research Method	7
4.1 Theoretical	7
4.2 Empirical Method Selection	7
4.2.1 Available Research Methodologies	7
4.2.2 Chosen Research Method	8
4.2.3 Experiment Design	8
4.2.4 Procedure	8
4.2.5 Control Measures	9
4.3 Data Collection Procedure	9
4.4 Operational Steps	10
4.4.1 RQ1: Relevant Test Cases	10
4.4.2 RQ2: False Positive	12
4.4.3 RQ3: Result Consistency	12
5 Results and Analysis	13
5.1 Data Analysis Procedure	13
5.1.1 RQ1	13
5.1.2 RQ2	15
5.1.3 RQ3	15
5.2 Results and Analysis	16
5.2.1 Answering RQ1	16
5.2.2 Answering RQ1.1	17
5.2.3 Answering RQ1.2	20
5.2.4 Answering RQ2	20
5.2.5 Answering RQ3	24

6	Discussion and Validity Threats	25
6.1	Discussion	25
6.1.1	Relevance Score	25
6.1.2	Function Characteristics and Relevance Score	25
6.1.3	False Positive Rate	26
6.1.4	Consistency of Test Execution	26
6.1.5	Our Study’s Limitations	27
6.2	Validity Threats	27
6.2.1	Internal Validity	27
6.2.2	External Validity:	28
7	Conclusion and Future Work	29
7.1	Conclusion	29
7.2	Future Work	29
7.2.1	Exploring Additional Functionalities:	29
7.2.2	Larger Sample Size:	30
7.2.3	Trustworthiness Across Programming Languages:	30
7.2.4	False Negative Rate Evaluation:	30
	References	31
A	Results from Codiumate	33
A.1	RQ1: Sorted Relevance Score from Scenario One.	34
A.2	RQ1.2 Relevance Scores From Scenario One and Scenario Two.	36
A.3	RQ2: False Positive Distribution Among Groups	37
A.4	RQ3: Result Consistency	39
B	Sample Functions, Selection Criteria, and Relevant Test Cases	40
B.1	Sample 1	40
B.2	Sample 2	43
B.3	Sample 3	45
B.4	Sample 4	48
B.5	Sample 5	50
B.6	Sample 6	52
B.7	Sample 7	54
B.8	Sample 8	56
B.9	Sample 9	58
B.10	Sample 10	61
B.11	Sample 11	63
B.12	Sample 12	65
B.13	Sample 13	68
B.14	Sample 14	70
B.15	Sample 15	73
B.16	Sample 16	76
B.17	Sample 17	80
B.18	Sample 18	83
B.19	Sample 19	90
B.20	Sample 20	94
B.21	Sample 21	98

B.22 Sample 22	100
B.23 Sample 23	104
B.24 Sample 24	107
B.25 Sample 25	110
B.26 Sample 26	114
B.27 Sample 27	117
B.28 Sample 28	120
B.29 Sample 29	123
B.30 Sample 30	126

In this chapter, you will gain an overview of the importance of unit testing, the challenges it presents, the current solutions available, and the existing research gaps in this field. Additionally, this chapter will outline the aim and objectives of this study, followed by a discussion on its scope.

Traditionally, the design and implementation of unit tests have been time-consuming and monotonous [7]. Thus transforming functional testing from a valuable opportunity for quality enhancement into a cumbersome task [11] [19]. This led to the development of automated unit test generation tools, such as EvoSuite [10], Randoop [16], JTEExpert [18] which primarily serve the Java programming language. However, these tools are typically restricted to specific programming languages and are constrained by their predefined algorithms [11].

Many innovative tools like ChatGPT and GitHub Copilot, based on OpenAI's Generative Pretrained Transformer (GPT) 3.5 and 4 large language models, have been introduced in the field of generative artificial intelligence, offering a new approach to test case design and test code generation offering the same quality as traditional automated testing tools [11].

Despite their advantages in enhancing the efficiency and reliability of software development, generative AI based tools, ChatGPT [11] and GitHub Copilot [22], encounter limitations of their own. Being general-purpose tools, they are not specifically designed for testing, and the quality of the generated unit tests depends on the user's manual input prompts. Additionally, using these tools may involve operational inconveniences, such as integrating code with OpenAI's application programming interface (API), incurring costs for accessing GitHub Copilot, the process of copying code from your code editor to ChatGPT and back to get the test codes.

1.1 Background

CodiumAI has developed a free to use Integrated Development Environment (IDE) plugin named Codiumate which aims to unlock the power of OpenAI's GPT-3.5/4 models to streamline the process of code behavior analysis (unit test case design), test code generation and evaluation, it provides an all-in-one environment for minimizing the repetitive task of designing, writing test code, and executing them. This tool is interesting because it is developed specifically for testing, it is free to use and convenient to use; it can be used directly in text editors, such as, Visual Studio Code (VS Code) and JetBrains, according to CodiumAI, the company behind Codiumate [8]. The statement in the previous sentence is not our ours; it is what CodiumAI claims, which sparked our interest

in evaluating the tool further.

Our goal is to investigate the trustworthiness of unit tests generated by CodiumAI for the Python programming language. We evaluate factors like test quality, reliability, and alignment with software requirements to provide insights into the effectiveness and limitations of Generative AI-based unit testing tools.

1.1.1 Trustworthiness in the Context of this Thesis

Trustworthiness refers to the degree of confidence that developers can have in the AI-generated unit tests. It encompasses the accuracy, reliability, and stability of the generated test cases, ensuring that they can be effectively integrated into the software development life-cycle without introducing errors or inconsistencies [12].

Accuracy (Relevance Score): Accuracy is measured by the Relevance Score, which assesses how well the generated test cases align with the expected behavior of the functions. This metric ensures that the test cases are appropriate and meaningful, directly reflecting the functional requirements [3].

Reliability (False Positive Rate): Reliability is captured by the False Positive Rate, indicating how often the generated test cases incorrectly report defects. A low false positive rate means that the tool is dependable and does not generate misleading error reports, which is crucial for maintaining developer trust and minimizing unnecessary debugging efforts [1].

Stability (Result Consistency Rate): Stability is evaluated through the Result Consistency Rate, which measures the consistency of test outcomes across multiple executions. High stability ensures that the generated tests are reliable over time and under various conditions, supporting their integration into automated testing processes [4].

1.2 Contribution

This study contributes to understanding the role of Generative AI in software testing in the context of Python programming language using Codiumate for unit testing.

1.3 Scope

Evaluation of Codiumate's Code Behavior Analysis (Test Case Suggestions):

- Analyzing the relevance of test cases suggested by Codiumate based on behavior coverage derived from function docstring and code implementation.
- Investigating the average Relevance Score of Codiumate's suggested test cases across various functions.

Factors Affecting Test Case Relevance:

- Examining the impact of different function characteristics, including function type, number of parameters, data types of parameters, average lines of code, and function return type, on the Relevance Score of suggested test cases.

Comparison with Standardized Test Cases:

- Comparing the Relevance Score of Codiumate's suggested test cases strictly to the selection criteria manually crafted by the authors.

Assessment of False Positive Results:

- Determining the frequency of false positive results in unit tests generated by Codiumate to assess its reliability.

Consistency Analysis:

- Evaluating the consistency of outcomes produced by the generated code for unit tests across multiple test executions to gauge the stability of Codiumate's testing approach.

In this chapter, you will be introduced to the research questions and definitions guiding our evaluation of Codiumate’s test case generation.

RQ1: To what degree are the test cases suggested by Codiumate relevant to function requirements?

Definition of Test Case: Test cases are defined as specific inputs and procedures applied during software testing to verify the software’s behavior [17]. The importance of selecting appropriate test cases is paramount, as noted by Patton [17] emphasizes that the choice of test cases is a critical task in software testing, influencing the effectiveness and efficiency of the testing process, moreover, improper selection can result in testing too much, testing too little, or testing the wrong things.

Definition of Relevant Test Case: The details of a test case should explain exactly what values or conditions will be sent to the software and what result is expected [17]. It exercises a particular program path or to verify compliance with a specific requirement [5]. The relevance of a test case in this study is the extent to which the suggested test cases by Codiumate align with and accurately reflect the expected behavior described in the selection criteria that were generated by an author. In the same way, all the selection criteria were based on the docstring of the functions under test, see an example in 4.2. The function docstring represents the function’s requirement. From these, the selection criteria will help both authors to validate and evaluate the suggested test cases generated by Codiumate.

RQ1.1: Which function characteristics have an impact on the relevance of test cases?

To grasp which factors impact the relevance of generated test cases by Codiumate, it’s essential to analyze how attributes or characteristics of a function influence Codiumate’s effectiveness in test case generation. By evaluating function characteristics such as function types (see details in 5.1, number of parameters and its type, average lines of code, and the return data type. From these, we might be able to identify if there are any relationships to the Relevance Scores, as it is not guaranteed and may require further investigation or analysis.

A Relevance Score, acts as a metric used to assess how well the model’s results align with the query. It’s a numerical value that reflects the extent to which the retrieved information is applicable or pertinent to the user’s request [9].

RQ1.2: How does the Relevance Score for test cases suggested by Codiumate vary compared to test cases from the selection criteria crafted by a developer?

Based on the test cases crafted by software developers, this research question aims to evaluate the trustworthiness and relevance of the test cases generated by Codiumate. By comparing the Relevance Scores, we can determine how well Codiumate's suggested test cases align with the expectations and requirements of developers

RQ2: What are the primary causes and implications of false positive results in unit tests generated by Codiumate?

Definition of False Positive: It is a test result in which a defect is reported, although no such defect exists in the test object [5]. In another word, a false positive is when a test wrongly shows there is an error in the code even though the code works fine. Many false positives have an impact on software reliability and code quality [6], which can reflex to the reliability of Codiumate test case generation as well. By finding out the false positive rates can help us evaluate the trustworthiness of Codiumate.

Definition of Unit Test: The testing of an individual software component [13]. Module testing (or unit testing) is a process of testing the individual subprograms, subroutines, classes, or procedures in a program [15]. In this study, unit tests are represented functions in each relevant test case generated by Codiumate.

This research question not only seeks to quantify the frequency of false positives but also aims to investigate the underlying reasons for these false positives and their impact on the development process.

RQ3: How consistent are the outcomes produced by the generated code for unit tests across multiple test executions?

Definition of Result Consistency: The use of consistency of test case results able to express the stability of features or functionalities provided or offered in the software under test [20]. Here, we can evaluate the stability attribute from the test outcomes generated by Codiumate, emphasizing the importance of producing identical results from the same test across multiple execution.

In recent years, there has been growing interest in the utilization of AI-driven tools for code generation and software development tasks. GitHub Copilot, introduced as an AI pair-programmer, has garnered attention for its ability to generate code snippets based on context and user input. Our research is grounded in the foundational research conducted by Yetistiren et al. [22], they examined the capabilities of GitHub Copilot in code generation tasks in terms of validity, correctness, and efficiency. They found that GitHub Copilot was able to generate valid code with a 91.5% success rate. For the correctness, out of 164 problems, 28.7% were correctly generated, 51.2% were partially correctly generated, and 20.1% were incorrectly generated [22].

While their study primarily scrutinized the quality of code snippets produced by Copilot, we draw upon their insights as a foundation to explore a complementary yet distinct facet of AI-driven development: unit test case generation.

Then another similar case study by Zhiqiang et al., conducted an empirical study to evaluate ChatGPT's capability of unit test generation [23]. They found that the tests generated by ChatGPT still suffer from correctness issues, including diverse compilation errors and execution failures. Even though, they were encouraged by that generating unit tests with ChatGPT could be very promising if the correctness of its generated tests could be further improved. Inspired by their findings above, they propose ChatTESTER, a novel ChatGPT-based unit test generation approach, which leverages ChatGPT itself to improve the quality of its generated tests. They focused on the quality of its generated tests regarding the correctness, sufficiency, readability, and usability. Their findings in effectiveness of ChatTESTER was that it generated 34.3% more compilable tests and 18.7% more tests with correct assertions than the default ChatGPT.

In this chapter, you will get an understanding of our research methodology for assessing the trustworthiness of Codiumate’s test case generation. Our empirical investigation involves data collection from various sources, including BTH’s library and online databases. We also detail our approach to gathering and evaluating data used to answer our research questions.

4.1 Theoretical

This research study was produced by gathering resources including literature reviews, previous studies, and other sources of information in the context of this field. References and citations for this research were gathered from below sources:

- BTH’s library
- Summon@BTH
- Scopus
- IEEE Xplore
- O’Reily
- Diva-portal
- Online articles

The keywords for the searching process are: AI, OpenAI, ChatGPT, Unit test, Test case, Codiumate, CodiumAI, Copilot, Trustworthiness.

Most reference findings were from Scopus. Each keyword was used in both singular and in combination in the searching process to find the relevance of ones.

4.2 Empirical Method Selection

4.2.1 Available Research Methodologies

In software engineering research, various methodologies are used to investigate phenomena and derive insights from. Here’s a summary of some common research methodologies [21]:

Experimentation: This method involves investigating testable hypotheses by manipulating independent variables to measure their effects on dependent variables. Typically conducted in a limited scope, often in a lab setting, and with high degree of control.

Survey: Surveys collect information to describe, compare, or explain knowledge, attitudes, and behavior of a population. Usually, surveys involve gathering data through interviews or questionnaires from a representative sample. Analyzing survey results yields descriptive and explanatory conclusions generalized to the population.

Case Study: Case studies investigate contemporary software engineering phenomena within real-life contexts. They draw on multiple sources of evidence to research specific projects, activities, or assignments. Data collection throughout the study enables statistical analyses aimed at tracking attributes or establishing relationships. Case studies offer lower control compared to experiments, but provide valuable insights into real-world scenarios.

4.2.2 Chosen Research Method

The research method chosen for this study is experimentation. This approach systematically evaluates Codiumate’s trustworthiness in generating unit tests by manipulating independent variables and observing their effects on the dependent variables.

4.2.3 Experiment Design

Independent Variables:

1. *Function Characteristics:* These include the type of function (e.g., mathematical operations, data manipulation), the number of parameters, and the data types of the parameters.
2. *Test Case Generation Tool:* The usage of Codiumate as the primary tool for generating unit tests.

Dependent Variables:

1. *Relevance Score:* This measures how well the generated test cases align with the expected behavior of the functions under test.
2. *False Positive Rate:* This indicates the frequency with which the generated test cases incorrectly identify defects.
3. *Result Consistency Rate:* This evaluates the consistency of test outcomes across multiple executions.

4.2.4 Procedure

1. **Selection of Functions:** Thirty functions were randomly selected from OpenAI’s HumanEval dataset. This random selection helps in minimizing selection bias and ensures a diverse range of functions are tested.

2. **Criteria Establishment:** Selection criteria for relevant test cases were established based on the functions' doc strings and expected behaviors.
3. **Test Case Generation:** Codiumate was used to generate unit tests for the selected functions. The generated test cases were then evaluated against the established criteria.
4. **Measurement and Analysis:**
 - (a) *Relevance Score:* Calculated by comparing the number of relevant test cases suggested by Codiumate to the total number of suggested test cases.
 - (b) *False Positive Rate:* Determined by executing the test cases and identifying how many of the failed tests were false positives.
 - (c) *Result Consistency Rate:* Evaluated by running the tests multiple times and measuring the consistency of the outcomes.

4.2.5 Control Measures

To ensure the accuracy and reliability of the results, several control measures were implemented:

1. **Consistent Evaluation Criteria:** A standardized set of criteria was used to evaluate the relevance of the test cases, ensuring uniformity in judgment.
2. **Multiple Test Runs:** Tests were run multiple times to check for consistency in results, thereby enhancing the robustness of the findings.

This case study aims to evaluate the relevance of test cases from two scenarios:

Scenario one: relates to RQ1, when the authors select the relevant test cases only in text form suggested by Codiumate based on the function under test requirement, see 5.1.1.1.

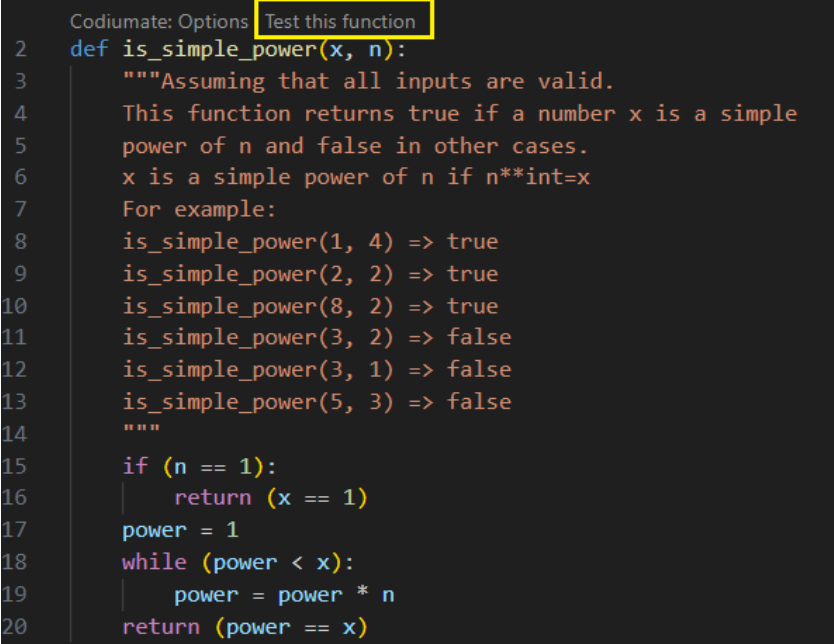
Scenario two: relates to RQ1.2, when the authors strictly evaluate all assertions within the selected test cases from the scenario one to see how many assertions matches the expected test cases from selection criteria, see 5.1.1.2.

4.3 Data Collection Procedure

Preparation: To evaluate Codiumate's trustworthiness, we follow the following procedure where step one, two and three below are executed once.

Step 1. Extract functions and tests: We extract functions and their corresponding unit tests from the HumanEval dataset [22] and store the result in our paper's GitHub repository.

Step 2. Randomly select 30 functions: Utilizing an online random number generator, we randomly generated 30 unique numbers between 0 and 163. These numbers correspond to the functions in the HumanEval dataset, which we used for our study. The choice of 30 functions was based on our pilot test run of the data collection's time constraints, where we found it took approximately 30 minutes to test and gather data for each function.



```

Codiumate: Options Test this function
2 def is_simple_power(x, n):
3     """Assuming that all inputs are valid.
4     This function returns true if a number x is a simple
5     power of n and false in other cases.
6     x is a simple power of n if n**int=x
7     For example:
8     is_simple_power(1, 4) => true
9     is_simple_power(2, 2) => true
10    is_simple_power(8, 2) => true
11    is_simple_power(3, 2) => false
12    is_simple_power(3, 1) => false
13    is_simple_power(5, 3) => false
14    """
15    if (n == 1):
16        return (x == 1)
17    power = 1
18    while (power < x):
19        power = power * n
20    return (power == x)

```

Figure 4.1: Codiumate test feature is above the sample function.

Step 3. Write Unit Test Selection Criteria: After the selection of 30 functions from HumanEval dataset, we wrote selection criteria which includes which unit tests are required to verify the functionality of the functions based on their docstring of each function as baseline. The selection criteria for each function from HumanEval dataset will be used as a filter to decide if a suggested test case by Codiumate is relevant. See a sample function with docstring in figure 4.1 and a sample of selection criteria in figure 4.2. Here, the test case selection process was performed by both authors.

4.4 Operational Steps

In the operational steps outlined below, outlines the steps we take to gather the data for answering RQ1, RQ2, and RQ3 respectively.

4.4.1 RQ1: Relevant Test Cases

Step 4. Understanding the function: We start by reading the function’s documentation (docstring) and the README.md file as selection criteria. This helps us understand what the function should do and how it should behave.

Step 5. Creating test cases: Using Codiumate’s **Test this function** feature, see figure 4.1, we automatically generate different scenarios to test the function.

Step 6. Evaluating the tests cases:

- We record the total number of test cases suggested by Codiumate, see figure 4.3 under **Total behaviors** (yellow circled).
- From **Behavior coverage** section, which contains the suggested test cases, we select the relevant test cases by clicking on the **Generate test** on the right side of the line in figure 4.3 (green circled). We do this by comparing them to our selection criteria.

```

1  # {#13} Test Case Selection Criteria for Function 76
2
3  ## Test Cases for `is_simple_power` Function
4
5  ### Boundary Values:
6  Test with boundary values where `x` and `n` are minimal or maximal.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1 | `1, 1` | `True` |
11 | 1 | `2, 2` | `True` |
12 | 2 | `1, 2` | `True` |
13 | 3 | `2, 1` | `False` |
14 | 4 | `2, 3` | `False` |
15
16 ### Typical Values:
17 Test with typical inputs where `x` is raised to the power of `n` and vice versa.
18
19 | Test Case | Input | Expected Output |
20 |-----|-----|-----|
21 | 5 | `8, 2` | `True` |
22 | 5 | `9, 3` | `True` |
23 | 5 | `16, 2` | `True` |
24 | 6 | `3, 2` | `False` |
25 | 6 | `5, 3` | `False` |
26
27
28 ### Special Cases:
29 Consider special cases mentioned or implied in the docstring. For instance:
30 - Test with `x` and `n` both being 0.
31
32 | Test Case | Input | Expected Output |
33 |-----|-----|-----|
34 | 7 | `0, 0` | `True` |
35
36

```

Figure 4.2: Selection Criteria for relevant test cases.

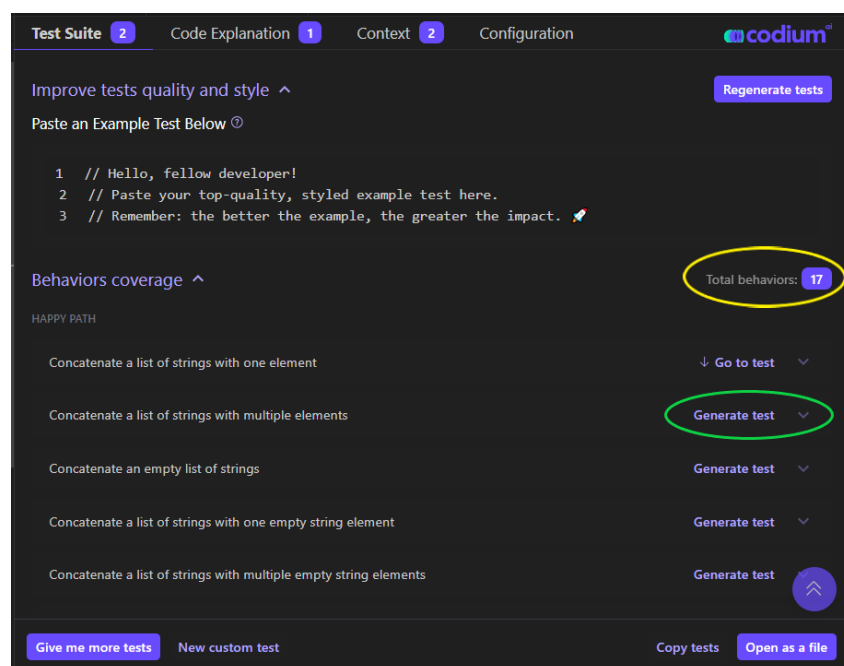


Figure 4.3: Codiumate plugin feature in IDE.

- Finally, We record the number of relevant test cases suggest by Codiumate.

Measuring Success: We calculate Relevance Score (RS), for RQ1, to get the number of relevance test cases were selected from the total suggested test cases by Codiumate. This score is expressed as a percentage using the following formula:

$$\bullet \text{ RS} = \left(\frac{\text{Number of relevant test cases from Codiumate}}{\text{Total number of test cases suggested by Codiumate}} \right) \times 100$$

4.4.2 RQ2: False Positive

Step 7. Generate Code:

- We proceed by generating code for the relevant test cases identified in RQ1.
- The process includes placing the test code into a separate file, following the naming convention: `function_<number>_test.py`. This file is in the *experiment result* directory.

Step 8. Evaluate the Generated Code for False Positive Result:

- We execute the tests and closely investigate the failed tests cases to check if they are due to false positive rate (FPR).
- We note the following on the side:
 - Time spent on data collection process for every selected sample:
 - Failed Unit Tests
 - Passed Unit Tests

Measuring Success: False positive rate is expressed as a percentage using the following formula:

$$\bullet \text{ FPR} = \left(\frac{\text{Number of false positive results}}{\text{Total number of relevant tests suggested by Codiumate}} \right) \times 100$$

4.4.3 RQ3: Result Consistency

Step 9. Get Result Consistency: We execute the tests five times and observe if they produce the same result. We then note both the number of changes, and the total test runs to calculate the result consistency rate.

Measuring Success: Result Consistency Rate (RCR) is expressed as a percentage using the following formula:

$$\bullet \text{ RCR} = \left(\frac{\text{Total test runs} - \text{number of changes}}{\text{Total test runs}} \right) \times 100$$

In the first section of this chapter, you will get an overview of our analysis procedures we applied to obtain the answers to our research questions, followed by the second section which answers the research questions stated in this paper.

5.1 Data Analysis Procedure

5.1.1 RQ1

Research question 1:

To what degree are the test cases suggested by Codiumate relevant to function requirements?

We employ the formula outlined in 4.4.1 to calculate the Relevance Score, which serves as our baseline for addressing the sub-questions for RQ1.

5.1.1.1 RQ1.1

Research question 1.1:

Which function characteristics have an impact on the relevance of test cases?

We sorted the collected data, see appendix A.2, in ascending order based on Relevance Score and then divide them into four groups ranging from:

Group	Relevance Score (RC) Range
1	$0 \leq RC \leq 25$
2	$25 < RC \leq 50$
3	$50 < RC \leq 75$
4	$75 < RC \leq 100$

Table 5.1: Sorted groups of sample functions based on Relevance Scores.

This approach allows us to analyze any similarities among functions within these groups and determine if the function characteristics introduced in RQ1.1 in chapter 2 impact the Relevance Score of the suggested test cases. By doing so, we aim to gain insights into whether Codiumate tends to suggest more relevant test cases for certain types of functions.

Function ID is a number that we have assigned to each function from 1 to 30 whereas Human Eval ID is the number of actual function in HumanEval dataset. These two IDs act as a mapping tool to understand which the relationship of each function in appendix A.2.

5.1.1.2 RQ1.2

Research question 1.2:

How does the Relevance Score for test cases suggested by Codiumate vary compared to test cases from the selection criteria crafted by a developer?

Even if a specific test case doesn't precisely fit our selection criteria, we may still consider it relevant. To ensure that our experience and bias haven't significantly skewed the count of relevant test cases, we standardized the Relevance Score. This means that only test cases containing assertions that match those listed in our selection criteria are deemed relevant. We followed a structured approach to standardize the Relevance Score. Firstly, in the selection criteria, we arranged relevant test cases in numerical order and categorized them into boundary types (based on boundary value analysis), typical values (those usually expected as function arguments), and special test cases reflecting various input-output scenarios. For instance, consider a function that takes a list of strings and a substring as arguments, returning true if the substring exists in the list and false otherwise. According to our criteria, a relevant test case would look like this:

Relevant Test Case:

- Input: (['car', 'boat', 'plane'], 'plane')
- Output: True
- Explanation: True is returned because 'plane' exists within the list of strings.

Even if a test case doesn't perfectly match the input-output specifics listed above but falls within the same category and yields a similar expected output, it is still considered relevant. For example:

Suggested Test Case:

- Input: (['cat', 'dog', 'fish'], 'fish')
- Output: True
- Explanation: The output is true since 'fish' exists within the list of strings.

Moving on to the `function_test_files` where we store the Pytest code for relevant test cases, we cross-reference each unit test against our selection criteria. We then tally the number of matches, disregarding duplicate cases within the same category. The Relevance Score is calculated by dividing the count of relevant test cases by the total number of expected test cases

5.1.2 RQ2

Research question 2:

What are the primary causes and implications of false positive results in unit tests generated by Codiummate?

To calculate the False Positive Rate (FPR), first, we arranged the collected data, see appendix A.3, in ascending order based on the Metric Relevance Score Column from RQ1. This ensured consistency in the order of functions between RQ1 and RQ2, matching the Function ID and Human Eval ID.

Second, we segmented the sorted data into four groups, following the exact division in 5.1.1.1. This segmentation allowed us to analyze if any group exhibited a notably higher False Positive Rate (FPR) and if there were any relationship between the groups identified in 5.1.1.1.

Third, we examined the relationship between false positives and various function characteristics that are used in RQ1.1, being function types, the number of parameters, parameter types, return types, and lines of code. This analysis involved investigating the impact and relationships among each function characteristic and its complexity toward the false positive rates.

Function ID is a number that we have assigned to each function from 1 to 30 whereas Human Eval ID is the number of actual function in HumanEval dataset. These two IDs act as a mapping tool to understand which the relationship of each function in appendix A.3.

5.1.3 RQ3

Research question 3:

How consistent are the outcomes produced by the generated code for unit tests across multiple test executions?

To calculate the Result Consistency Rate (RCR), we sorted the collected data, see appendix A.4, in ascending order based on the Metric Relevance Score Column from RQ1. This ensured alignment between the function order in RQ3 and RQ1, preserving consistency in Function ID and Human Eval ID.

Subsequently, we divided the sorted data into four groups, following the exact division in 5.1.1.1. based on the relevance scores derived in RQ1. This segmentation allowed us to examine if any of the groups exhibited inconsistency in results and whether there were any relationships between the relevance score.

Function ID is a number that we have assigned to each function from 1 to 30 whereas Human Eval ID is the number of actual function in HumanEval dataset. These two IDs act as a mapping tool to understand which the relationship of each function in appendix A.3.

5.2 Results and Analysis

5.2.1 Answering RQ1

The Relevance Score for the suggested test cases under behavior coverage by Codiumate has an average of 47.97%. This score varies from a minimum of 10.53% to 100%.

The average Relevance Score of 47.97% indicates that, on average, slightly less than half of the test cases generated by Codiumate are relevant to the function requirements. This suggests that while Codiumate is capable of generating some relevant test cases, there is significant room for improvement in its ability to consistently produce test cases that meet the function requirements.

The wide range in scores, from as low as 10.53% to as high as 100%, indicates variability in Codiumate’s performance. In some instances, Codiumate is highly effective, producing entirely relevant test cases (100% relevance). However, in other cases, it performs poorly, with a low relevance score of 10.53%. This variability suggests that the effectiveness of Codiumate’s test case generation may depend on specific factors such as the complexity of the function, the characteristics of the function, and the clarity of the function requirements.

Overall, these findings highlight the potential of Codiumate to assist in generating relevant test cases, but also underscore the need for further refinement and enhancement to improve its reliability and consistency in generating functionally relevant test cases.

In Scenario One, we calculated the Relevance Score using the formula outlined in section 4.4.1. in Figure 5.1, the mean Relevance Score is highlighted by the red rectangle, while the smallest Relevance Score’s range is marked by the yellow rectangle, and the largest Relevance Score’s range is indicated by the green rectangle.

Percentiles		Smallest		
1%	10.52632	10.52632		
5%	13.33333	13.33333		
10%	14.16667	13.33333		
25%	21.42857	15	Obs	30
			Sum of wgt.	30
50%	39.23077		Mean	47.97097
75%	72.22222	Largest	Std. dev.	28.03011
90%	91.66667	90	Variance	785.6871
95%	100	93.33334	Skewness	.4652659
99%	100	100	Kurtosis	1.975823

Figure 5.1: Scenario One, Relevance Score.

In Scenario Two, we examined the relevant test cases selected from Codiumate. Our approach strictly adhered to the predefined selection criteria. This step was crucial to ensure that our personal experience didn’t influence the selection of relevant test cases from Codiumate. Furthermore, Scenario Two aimed to validate the results obtained in Scenario One, providing confidence in the accuracy and reliability of our findings, see figure 5.2.

Relevance_score				
	Percentiles	Smallest		
1%	40	40		
5%	42.86	42.86		
10%	42.86	42.86	Obs	30
25%	55.56	42.86	Sum of wgt.	30
50%	71.43		Mean	70.44167
75%	85.71	Largest	Std. dev.	18.65674
90%	100	100	Variance	348.0741
95%	100	100	Skewness	.0227679
99%	100	100	Kurtosis	1.927567

Figure 5.2: Scenario Two, Relevance Score.

5.2.2 Answering RQ1.1

Among the five factors outlined in RQ1.1, only four factors—Function type, number of parameters, parameter data types, and function’s return type exerted an influence on the Relevance Score. Conversely, the average line of code (LOC) per group did not impact the Relevance Score.

5.2.2.1 Function Type:

Among the five function types, Math (Pure mathematics calculation), filtering (Filters Data without changing the original data), Evaluation (Evaluates if an input results to true or false), Data Manipulation (Change the original data), and Algorithm (Sort, search for data) Math exhibits a lower Relevance Score compared to Evaluation and Data Manipulation. Filtering and Algorithm function types show no discernible impact on the Relevance Score; there are no apparent patterns indicating their influence.

As illustrated in figure 5.3, 87.5% of functions in Group 1 belong to the Math type, whereas only 14.2% of functions in Group 4 are Math-type functions. This trend suggests that Codiumate’s performance is weaker for Math-type functions as the Relevance Score increases.

The lower Relevance Score for Math functions suggests that Codiumate may struggle with generating relevant test cases for pure mathematical calculations. This might be due to the abstract and complex nature of mathematical functions.

In contrast, there is no representation of Filtering functions in Group 1, nor in Group 4, indicating insufficient evidence to assess Codiumate’s performance for this function type; no noticeable trend is observed.

For Evaluation function types, a clear upward trend is evident: the percentage of Evaluation functions increases with higher Relevance Scores. For instance, no Evaluation functions are present in Group 1, while 42.8% of functions in Group 4 are Evaluation types, suggesting better performance by Codiumate for Evaluation functions.

Similarly, Data Manipulation function types also exhibit an upward trend: as the Relevance Score rises, so does the percentage of Data Manipulation functions. For example, no Data Manipulation functions are found in Group 1, while 36.5% of functions in Group 4 fall into this category, indicating improved performance for Data Manipulation functions by Codiumate.

Conversely, there is no discernible trend for Algorithm function types. For instance, 12.5% of all functions in Group 1 are Algorithm types, while 12.2% of functions in Group 4 are of this type. This lack of consistent trend suggests insufficient evidence to evaluate Codiumate’s performance for Algorithm functions. A larger sample size could have provided more insight into Codiumate’s performance for Algorithm function types.

Group Name	Function Type: Math (%)	Function Type: Filtering (%)	Function Type: Evaluation (%)	Function Type: Data Manipulation (%)	Function Type: Algorithm (%)
Group 1	87.5	0	0	0	12.5
Group 2	44.4	33.3	11.1	11.1	0
Group 3	16.6	33.3	33.3	16.6	0
Group 4	14.2	0	42.8	28.5	14.2

Figure 5.3: Relationship between Relevance Score in sample groups and function characteristics.

Analysis for function type:

5.2.2.2 Number of Parameters:

There is a weak relationship between the Relevance Score and the number of parameters, as evidenced in figure 5.4 below. In Group 1, for instance, 62.4% of functions had two parameters, whereas only 14.2% had two parameters in Group 4. Conversely, in Group 1, only 37.5% of functions had one parameter, while in Group 4, 85.7% had one parameter. This trend indicates that as the number of parameters decreases, the Relevance Score may increase, but due to limited evidence we cannot conclude that the relevance score is impacted by the number of parameters.

Group	Function Parameters: 1 (%)	Function Parameters: 2 (%)	Function Parameters: 3 (%)
Group 1	37.5	62.5	0
Group 2	77.7	22.2	0
Group 3	66.6	16.6	16.6
Group 4	85.7	14.2	0

Figure 5.4: Relationships between sample groups and function with number of parameters 1, 2, and 3 respectively.

5.2.2.3 Data Type of Parameter:

Among the three parameter data types, integer, string, and float. Integer exhibits a lower Relevance Score compared to string and float data types.

As observed in figure 5.5, there is a downward trend in the percentage of integer data types as the Relevance Score increases. For instance, 76.9% of total parameter data types in Group 1 are integers, while only 37.5% of parameter data types in Group 4 are integers. This trend suggests that Codiumate’s performance is weaker for functions with integer-type parameters.

In contrast, there is an upward trend for string parameter types: as the Relevance Score increases, so does the percentage of string-type parameters. However, there is a 25% downward trend, indicating some variability. This suggests that Codiumate may perform better for functions with string-type parameters, although there is some uncertainty.

Similarly, there is an upward trend for float parameter types: as the Relevance Score increases, the percentage of float type parameters also increases. However, since no float-type parameters are present in groups One and Three, a consistent upward trend

is not observed. A larger sample size could have provided more insight into Codimate's performance for float type parameters.

Group Name	Function Parameters Data Type: Integer (%)	Function Parameters Data Type: String (%)	Function Parameters Data Type: Float (%)
Group 1	76.9	23.1	0
Group 2	45.4	45.4	9
Group 3	44.4	55.5	0
Group 4	37.5	37.5	25

Figure 5.5: Relationships between sample groups and function input types.

5.2.2.4 Average Lines of Code (LOC) per group:

There is no clear and consistent trend indicating whether the average LOC per group has a significant impact on the Relevance Score. For instance, in figure 5.6 below, Group 1, which has the lowest Relevance Score, has an average LOC of six lines, while Group 2, with a higher Relevance Score, has an average LOC of 3.5 lines. One might initially assume that the average LOC decreases as the Relevance Score increases. However, in groups three and four, the average LOC per group increases.

This lack of consistent trend suggests that there is not enough compelling evidence to determine if the average LOC has any notable impact on the Relevance Score. In another word, the function characteristic that measured by the number of LOC does not significantly impact the relevance of generated test cases. This suggests that the Codiumate's ability to generate relevant test cases is more influenced by the nature and complexity of the inputs and outputs rather than the length of the code.

Group Name	Average Number of Code Lines
Group 1	6
Group 2	3.5
Group 3	4.6
Group 4	7.2

Figure 5.6: relationship between sample groups and average lines of code.

5.2.2.5 Function Return Data Type:

Among the four data return types (Integer, Boolean, String, and Float), there is only a slight indication that Codiumate performs better for String return data types and worse for Integer data types.

According to figure 5.7, 62.5% of the return types in Group 1 are Integer, while only 33.3% are in Group 2 and 3. One might assume that the Relevance Score increases as the percentage of Integer data type decreases, but in Group 4, which has the highest Relevance Score, 50% of the data types are Integer. This suggests that Codiumate might perform better for functions that produce data types other than integers.

There is no clear and consistent relationship between the Boolean return type and the Relevance Score. For example, 12.5% of the return types in Group 1 are Boolean, while 22.2% are in Group 2. While one might expect the percentage of Boolean data types to increase as the Relevance Score increases, in Group 3 and 4, it decreases back to 16.6%. This indicates that there is no relationship between the Relevance Score and Boolean return type.

Regarding the string return type, we observe a consistent upward trend from Group 1 to 3, with a drop-in Group 4. For example, 25% of the return types in Group 1 are String, while 50% are in Group 3. This suggests that as the percentage of String return type increases, so does the Relevance Score. However, in Group 4, which has the highest Relevance Score, the percentage drops to 25%, indicating that Codiumate may perform better for String return types.

Lastly, for the return type of Float, there is not enough data to draw any conclusions.

Percentage of Return Type Found				
Group Name	Integer	Boolean	String	Float
Group 1	62.5	12.5	25	0
Group 2	33.3	22.2	33.3	11.1
Group 3	33.3	16.6	50	0
Group 4	50	16.6	33.3	0

Figure 5.7: Relationships between sample groups and percentage of return types.

5.2.3 Answering RQ1.2

Upon zooming in and scrutinizing the relevance scores on average, we observe that in Scenario One, the mean relevance score across all groups was 47.97%. However, in Scenario Two, the mean relevance score increased significantly to 70.41%.

The significant increase in the mean relevance scores from Scenario One to Scenario Two across most groups suggests that the initial selection of test cases by Codiumate often lacks complete alignment with the expected assertions. However, once assertions are scrutinized and aligned more closely with developer expectations (Scenario Two), the relevance of the test cases improves markedly. This indicates that while Codiumate's automated suggestions are helpful, they often require further refinement and verification to meet the developers' standards.

For instance, in Group 1, the relevance score increased substantially from 16.88% to 58.39%, demonstrating a notable improvement when assertions are carefully evaluated. Similar patterns are observed in Groups 2 and 3, with relevance scores rising significantly upon detailed examination. Group 4 saw a slight decrease in relevance score from 88.84% to 79.24%, suggesting that for this group, the initial suggestions were closer to the expected outcomes, and further scrutiny identified some misalignment. see figure 5.8. See detailed results in the appendix A.2

Scenario 1: Group	Scenario 1: Mean-RC	Scenario 2: Mean-RC
1	16.88	58.39
2	36.13	69.54
3	59.50	77.60
4	88.84	79.24

Figure 5.8: Relevance score Means from Scenario One and Scenario Two based on sorted groups from RQ1.1.

5.2.4 Answering RQ2

The total count of false positives across all 208-unit tests is 31, which comprises 14.90% of the total relevant unit tests. Among the 42 failed tests, 31 are false positives, which

represents a false positive frequency of 73.81%. The average false positive rate (FPR) is 9.91%. These false positive rates vary from a minimum of 0% to a maximum of 69.23%. Figure 5.9 below shows that FPRs are higher in Group 3, 2, and 4 respectively, no false positives found in Group 1. More details are shown in appendix A.3.

To understand the primary causes and implications of false positive results in unit tests generated by Codiumate, we conducted a deeper analysis examining the relationship between FPRs and various function characteristics. These characteristics include function types, number of parameters, parameter data types, return types, and lines of code (LOC).

5.2.4.1 False Positives and Function Types

Figure 5.9 shows that Group 1 has no false positives. The sample functions are predominantly Math (87.5%) and Algorithm (12.5%).

Group 2 has FPR of 13.5%. This group includes a mix of function types, with a significant portion being Math (44.4%) and Filtering (33.3%)

Group 3 has the highest FPR of 15.7%. This group has a diverse set of function types, with Filtering and Evaluation each making up 33.3%.

Group 4 has FPR of 11.60%. This group is heavily weighted towards Evaluation (42.8%) and Data Manipulation (28.5%).

Math functions tend to have lower false positive rates. Group 1, which is predominantly Math functions, has no false positives. Filtering and Evaluation functions appear to be associated with higher false positive rates, especially when combined with other function types. Groups with a diverse set of function types (groups two, three, and four) show higher false positive rates, suggesting that complexity and diversity of function types might contribute to false positives. Group 4, with high Evaluation and Data Manipulation functions, has a relatively high FPR, indicating these types might be more prone to false positives

Group Name	FPRs	Function Type (%)				
		Math	Filtering	Evaluation	Data Manipulation	Algorithm
Group 1	0.0	87.5	0	0	0	12.5
Group 2	13.5	44.4	33.3	11.1	11.1	0
Group 3	15.7	16.6	33.3	33.3	16.6	0
Group 4	11.6	14.2	0	42.8	28.5	14.2

Figure 5.9: False Positive Rates and Function Types.

5.2.4.2 False Positives and Number of Parameters

Figure 5.10 shows that Group 1 has no false positives. The sample functions predominantly contain two parameters (62.5%) and one parameter (37.5%).

Group 2 has FPR of 13.5%. This group includes a mix of function types, with a significant portion being one parameter (77.4%) and two parameters (22.2%)

Group 3 has the highest FPR of 15.7%. This group has a diverse set of parameters, with the majority of sample functions with one parameter (66.6%), followed by sample functions with two and three parameters (16.6% and 16.6%).

Group 4 has FPR of 11.60%. This group is heavily weighted towards sample functions with one parameter (85.7%) and two parameters (14.2%).

Group Name	FPRs	Number of Function Parameter (%)		
		1	2	3
Group 1	0.0	37.5	62.5	0
Group 2	13.5	77.7	22.2	0
Group 3	15.7	66.6	16.6	16.6
Group 4	11.6	85.7	14.2	0

Figure 5.10: False Positive Rates and Number of Parameters.

Group 1 has the highest percentage of two parameter functions and the lowest FPR. One could interpret that functions with two parameters strike a balance between complexity and testability.

Groups with fewer two-parameter functions (Groups two, three, and four) have higher FPRs, reinforcing the notion that moderate complexity improves test case accuracy.

Group 3 is the only group with a significant percentage of three parameters functions and has the highest FPR. This indicates that functions with three parameters might introduce complexity that leads to higher false positives, possibly due to the increased difficulty in generating comprehensive test cases.

5.2.4.3 False Positives and Function Parameter Data Types

See figure 5.11 for the below analysis:

Integer parameter type; Group 1 contains functions with a higher percentage of integer parameters and have the lowest FPR (0.0%), indicating that integer parameters are easier to test accurately.

Lower percentages of integer parameters in other groups (Groups two, three, and four) correlate with higher FPRs, suggesting that the presence of integer parameters contributes positively to the reliability of generated test cases.

String parameter type; Higher percentages of string parameters are associated with higher FPRs (Groups two, three, and four). This indicates that functions with string parameters are more prone to false positives.

String parameters introduce complexity due to the variability in possible values, which might be challenging for Codiumate's test generation algorithms to handle effectively.

Float parameter type; The presence of float parameters in Group 2 (9%) and Group 4 (25%) correlates with higher FPRs. This may suggest that float parameters, due to their precision and rounding issues, are more difficult to test accurately.

Group 1 and Group 3 have no float parameters, yet Group 3 still has a high FPR, indicating that other factors (such as the presence of string parameters) also play a significant role.

Group Name	FPRs	Number of Parameter Data Type (%)		
		Integer	String	Float
Group 1	0.0	76.9	23.1	0
Group 2	13.5	45.4	45.4	9
Group 3	15.7	44.4	55.5	0
Group 4	11.6	37.5	37.5	25

Figure 5.11: False Positive Rates and Function Parameter Types.

5.2.4.4 False Positives and Return Data Types

Figure 5.12 shows the relationships between false positive rates and return types that can be interpreted as following:

Integer Return Type; functions with higher percentages of Integer return types, such as Group 1 (62.5%), have lower FPRs. This suggests that integer-based functions are easier to validate correctly with generated unit tests. Even in groups with moderate integer percentages, like Group 4 (50%), the FPRs remain relatively lower compared to other groups, indicating a trend of lower FPRs with integer return types.

Boolean Return Type; Boolean return types are less prevalent across all groups. However, the data does not show a clear pattern linking Boolean return types to lower or higher FPRs. This indicates that Boolean return types may not significantly influence the accuracy of generated tests. This suggests that functions returning strings are more prone to false positives.

String Return Type; higher percentages of String return types, particularly in Group 3 (50%), are associated with the highest FPRs. String handling can be complex due to variability in input and output, leading to less reliable test cases.

Float Return Type; Float return types are only present in Group 2 (11.1%), which has a relatively higher FPR of 13.5%. Although the sample size is small, the presence of float return types correlates with higher FPRs, indicating potential challenges in accurately testing floating-point operations.

Group Name	FPRs	Number of Return Type (%)			
		Integer	Boolean	String	Float
Group 1	0.0	62.5	12.5	25	0
Group 2	13.5	33.3	22.2	33.3	11.1
Group 3	15.7	33.3	16.6	50	0
Group 4	11.6	50	16.6	33.3	0

Figure 5.12: False Positive Rates and Return Type.

Functions with Integer handling associated with lower FPRs, indicating these functions are easier to test accurately. Higher percentages of String return types correlate with higher FPRs, suggesting challenges in testing functions with string outputs. Presence in Group 2 indicates higher FPRs, suggesting difficulties in testing floating-point operations.

5.2.4.5 False Positives and Average Lines of Code (LOC) of sample functions

From figure 5.13, Group 2 and three have higher FPRs for functions with shorter LOC indicate that Codiumate might be generating less effective tests for simpler functions. Short functions might lack sufficient complexity to generate meaningful test cases, leading to higher false positives.

Group 1 and Group 4 contain moderate to longer LOC, these groups have lower FPRs, this may suggest that Codiumate handles more complex functions better, producing more accurate and reliable test cases.

Group Name	FPRs	Average Lines of Code
Group 1	0.0	6
Group 2	13.5	3.5
Group 3	15.7	4.6
Group 4	11.6	7.2

Figure 5.13: False Positive Rates and Lines of Code from Function Under Test.

5.2.5 Answering RQ3

The result consistency for the code generated by Codiumate was 100% across all five executions, consistently producing the same result each time. However, it's important to note that since the test codes were not executed in Codiumate's runtime environment, this data does not contribute much to determining Codiumate's trustworthiness in its runtime environment.

We executed the test codes on our local machine using Pytest and the test code produced by Codiumate ran error-free, without any syntax errors. The error-free execution of the test code was coincidental rather than intentional because we did not execute within Codiumate's runtime environment but were run locally. Therefore, the observation that the test code ran without errors was unexpected and not part of our planned evaluation process.

We aimed to assess the trustworthiness of Codiumate through several key metrics: Relevance Score, false positive rate, and the consistency of test case outcomes. Our findings provide insights into the strengths and limitations of Generative AI-driven test generation tools like Codiumate, particularly in the context of their applicability to real-world software development from a developer’s point of view.

6.1 Discussion

6.1.1 Relevance Score

Initially, we observed an average Relevance Score of 48% for the test cases suggested by Codiumate in Scenario One, which we deemed acceptable for a first attempt. However, when we closely compared the selected test cases against our criteria in Scenario Two, the Relevance Score increased to 70%. This unexpected improvement indicated that Codiumate’s trustworthiness was higher than we initially thought. We realized that the process of selecting relevant test cases in Scenario One could have been more thorough by reviewing all assertions rather than just the test case descriptions provided by Codiumate. Despite this, the findings from Scenario Two suggest that Codiumate’s test case generation functionality is reliable. From a developer’s perspective, even a 48% Relevance Score is significant, as it shows that Codiumate can facilitate and simplify the unit testing process, even if it cannot fully automate it yet.

6.1.2 Function Characteristics and Relevance Score

After categorizing the sample functions into four groups based on their Relevance Scores, we observed a distinct pattern: the group with lower Relevance Scores contained a higher proportion of functions involving mathematical calculations, while conversely, groups with higher Relevance Scores exhibited fewer math-related functions. The intricacies of mathematical operations demand precise calculations, logical reasoning, and formal deduction, attributes that are not inherently strong suits of the GPT model architecture, including Codiumate powered by GPT-3.5 and 4 [2]. These models are primarily engineered to excel in processing and generating coherent, human-like text, rendering them more adept at tasks such as language translation or text generation. Consequently, our findings indicate that Relevance Scores tended to be higher for functions involving string inputs, aligning with the inherent strengths of the underlying model.

Our observations parallel those made by Jessica et al. [14], who examined the proficiency of GPT-3.5, GPT-4, and BARD models across various tasks, including mathematical and multi-hop reasoning. Their findings corroborate our own, suggesting that these

models exhibit limited proficiency in Mathematical, Inductive, and Multi-hop reasoning tasks. In essence, our research supports the notion that, at the time of our study, GPT-3.5 and 4 are still not adequately equipped to handle mathematical operations effectively.

6.1.3 False Positive Rate

Codiumate produced approximately 15% false positives out of all the relevant test cases. Upon closer inspection, we found that nearly 74% of failed tests were due to false positive result, often due to minor errors in the relevant test cases. These defects were generally insignificant. These defects were generally insignificant but indicate areas for improvement. Potential causes could be misconfigured test assertions or errors in the test logic itself. For example, Codiumate might generate assertions that do not align with the actual requirements of the function. Consequently, these test cases require supervision from developers, necessitating a basic understanding of unit tests and testing principles.

The maximum false positive rate observed was over 69%, which is significant and can severely impact the reliability and efficiency of the testing process. Developers might spend considerable time debugging non-existent issues, which can lead to reduced trust in the AI-automated testing tool and potentially lower overall productivity.

False positive rate and function complexity, functions with simpler structures (fewer lines of code and parameters) and those involving complex data types (strings and floats) tend to have higher false positive rates. This suggests that Codiumate's current algorithms may not handle these scenarios effectively, leading to a higher incidence of false positives.

While Codiumate demonstrates some reliability, particularly with functions of moderate complexity (e.g., those with math function type, integer parameters, and moderate lines of code), it is not completely trustworthy in its current state. The tool's tendency to produce false positives in certain scenarios suggests areas where improvements are needed: Codiumate needs to improve its handling of string and float parameters. These data types introduce variability and complexity that current algorithms struggle with, resulting in higher false positive rates. Better alignment of generated assertions with actual function requirements is crucial. Codiumate should focus on reducing misconfigured test assertions and logical errors in test cases to lower the incidence of false positives.

Despite its automation capabilities, Codiumate's test cases still require supervision and review from developers. Encouraging developers to manually review and supplement automatically generated tests can enhance the overall reliability and trustworthiness of the tool.

6.1.4 Consistency of Test Execution

The consistency of test execution using Pytest was 100%, with 42 (20%) failed tests and 166 (80%) passed tests, all without syntax errors across five executions, and without any intervention from the authors. This indicates that Codiumate generates correct and stable syntax. However, this consistency does not significantly impact real-world software development, since the sample functions used were standalone with simple syntax.

Our study also suggests that full test automation in the context of unit testing is not yet feasible. Developers still need to select relevant test cases and manually verify that failed test cases are not due to false positives.

6.1.5 Our Study’s Limitations

A limitation of our study is that we only investigated the false positive rate, covering 20% of the relevant test cases (the percentage of failed tests). We did not check the remaining 80% (the percentage of passed tests) for false negatives. Additionally, our sample data was limited to 30 Python functions, which means our findings may not apply to functions written in other programming languages.

The limitations above have significant implications for the generalizability and completeness of our findings. Firstly, by focusing only on the false positive rate, we have potentially overlooked a critical aspect of Codiumate’s performance: the false negative rate. False negatives—cases where defects are missed—can be just as detrimental to software quality as false positives, leading to undetected bugs and erroneous functionality. Without evaluating the 80% of passed tests for false negatives, our assessment of Codiumate’s reliability remains incomplete. This gap suggests that the tool might miss critical issues, which could compromise the integrity of the testing process.

Furthermore, the sample size and scope of our study limit the applicability of our results. Examining only 30 Python functions restricts the breadth of our conclusions, as this small dataset may not fully represent the diversity and complexity of real-world software. Different programming languages and paradigms could yield different outcomes, and Codiumate’s performance might vary significantly across various types of software applications and coding styles. As a result, our findings may not be fully applicable to functions written in other programming languages or in more complex software environments.

These limitations highlight the need for more comprehensive studies to validate Codiumate’s trustworthiness across diverse scenarios. Future research should include larger, more varied datasets and evaluate both false positives and false negatives. This would provide a more robust and holistic understanding of the tool’s capabilities and limitations, ultimately leading to more reliable and generalizable conclusions about its utility in automated unit testing.

6.2 Validity Threats

6.2.1 Internal Validity

6.2.1.1 Researchers’ Prior Experience in Testing:

As software engineering students, our experience and understanding of unit testing influence the selection of relevant test cases, potentially introducing bias. To mitigate this, we collaboratively evaluated the first 10 functions to standardize our approach, ensuring a consistent methodology for subsequent evaluations.

6.2.1.2 Researchers’ Data Collection Bias:

Since this process is performed by two authors independently, there is a risk of inter-rater variability in our evaluations of the trustworthiness of the generated test cases. Differences

in interpretation, judgment, or criteria for assessment could introduce bias and affect the reliability of the study results. To reduce this risk, we implemented measures to maintain objectivity, such as standardized procedures for test case evaluation and cross-checking each other's work.

6.2.1.3 Timing Discrepancies:

The timing of the testing sessions conducted by each author may vary due to factors such as workload, interruptions, or personal schedules. These timing discrepancies could influence the performance of Codiumate and the generated test cases, potentially leading to inconsistent outcomes.

6.2.2 External Validity:

6.2.2.1 Sample Size and Generalizability:

The study's use of 30 Python functions from the HumanEval dataset limits the generalizability of the findings. This sample may not represent the full diversity of real-world software development scenarios, potentially affecting the applicability of the results. Future studies should include a broader and more diverse set of functions to enhance the generalizability of the findings.

6.2.2.2 Testing Environment Variability:

In testing the execution process, both authors performed the tasks with an individual's computer. This differences in computer hardware, software configurations, and environmental factors between the two authors' testing environments could introduce variability in the results. This variability may impact the consistency of the findings.

6.2.2.3 Software Versioning and Updates:

Changes in the versions of Codiumate or its underlying GPT models (GPT-3.5 and 4) during the course of the study could impact the behavior and performance of the plugin. This could introduce confounding variables that affect the validity of the conclusions drawn from the study.

7.1 Conclusion

The evaluation of Codiumate’s test case generation highlights both its promise and areas requiring improvement. Trustworthiness, in the context of this thesis, refers to the degree of confidence developers can have in the AI-generated unit tests, encompassing accuracy, reliability, and stability.

Codiumate shows a good success rate in identifying relevant test cases, particularly for non-mathematical functions and less complex functions. However, developers must possess a fundamental understanding of unit testing principles and function behavior to effectively utilize and evaluate the generated test cases.

The overall false positive rate of approximately 15% indicates that while Codiumate’s test case generation is promising, it is not entirely trustworthy in its current form. Specific areas of concern include handling complex data types (e.g., strings and floats) and ensuring test assertions align with actual function requirements.

Codiumate generated unit tests are free of syntax errors and demonstrate dependable performance across various functions, particularly those that are not overly complex. However, the stability of test outcomes across multiple executions still requires thorough evaluation to ensure long-term reliability.

Codiumate demonstrates significant potential in automating unit test generation, offering a convenient means to support developers. However, it is not yet fully reliable for critical applications without developer oversight. Continued refinement and exploration of its capabilities are essential for Codiumate to become an indispensable asset in unit test generation, enhancing its trustworthiness and effectiveness in the software development process.

By understanding the underlying mechanisms powered by GPT-3.5 and GPT-4, developers can better appreciate the strengths and limitations of Codiumate, making informed decisions about its integration into their development workflows.

7.2 Future Work

7.2.1 Exploring Additional Functionalities:

Beyond behavior analysis and unit test code generation, Codiumate offers additional functionalities that are worth exploring. Investigating aspects such as the impact of integrating pre-written high-quality tests on test case quality can provide valuable insights into optimizing Codiumate’s functionalities for enhanced efficiency and effectiveness. Another aspect would be to generate tests for functions that rely on other functions to evaluate

if Codiumate's produced code for the unit test isolate the function; test the function in isolation.

7.2.2 Larger Sample Size:

In our study, we used 30 out of 164 functions from HumanEval dataset, which is very limited. Expanding the sample size to 164 or even beyond the HumanEval dataset can provide valuable insights into the strengths and limitations of Codiumate.

7.2.3 Trustworthiness Across Programming Languages:

Exploring Codiumate's trustworthiness in the context of different programming languages is essential for assessing its versatility and adaptability. By testing its performance in languages beyond the current scope, we can ascertain its reliability across diverse technological ecosystems, thereby enhancing its utility and applicability.

7.2.4 False Negative Rate Evaluation:

To fully assess Codiumate's effectiveness in defect detection, it's important to study and evaluate its false negative rate. By analyzing cases where Codiumate failed to identify defects in functions, we can evaluate its reliability in catching potential issues. This evaluation will provide crucial insights into the tool's overall performance and areas for improvement.

References

- [1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.
- [2] Baeldung, “Using chatgpt to solve math problems,” Year, accessed on 22-04-2024, (n.d.). [Online]. Available: <https://www.baeldung.com/cs/chatgpt-math-problems#:~:text=While%20ChatGPT%20may%20excel%20at,mechanism%20to%20perform%20mathematical%20computations>
- [3] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.
- [4] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *2007 Future of Software Engineering*. IEEE, 2007, pp. 85–103.
- [5] R. Black, E. V. Veenendaal, and D. Graham, *FOUNDATIONS OF SOFTWARE TESTING*, 3rd ed. Cengage Learning, Inc., 2018.
- [6] C. Boogerd and L. Moonen, “Prioritizing software inspection results using static profiling,” in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, 2006.
- [7] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, “Chatunitest: A framework for llm-based test generation,” 2024.
- [8] CodiumAI, 2024, accessed: 2024-06-13, (n.d.). [Online]. Available: <https://www.codium.ai/>
- [9] Deepset, “Relevance scores,” accessed: 12-05-2024, (n.d.). [Online]. Available: <https://docs.cloud.deepset.ai/docs/relevance-scores>
- [10] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419.
- [11] V. Guilherme and A. Vincenzi, “An initial investigation of chatgpt unit test generation capability.” Association for Computing Machinery, 9 2023, pp. 15–24.
- [12] M. Harman, Y. Jia, and Y. Zhang, “Achievements, open problems and challenges for search based software testing,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–12.
- [13] I. S. T. Q. B. (ISTQB), “Unit testing,” 2024, accessed: 2024-06-10, (n.d.). [Online]. Available: <https://istqb-glossary.page/unittesting/#:~:text=The%20testing%20of%20individual%20software%20components>

- [14] J. López Espejel, E. H. Ettifouri, M. S. Yahaya Alassan, E. M. Chouham, and W. Dahhane, “Gpt-3.5, gpt-4, or bard? evaluating llms reasoning ability in zero-shot setting and performance boosting through prompts,” *Natural Language Processing Journal*, vol. 5, p. 100032, 2023.
- [15] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The art of software testing*. John Wiley Sons, Inc., 2004.
- [16] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for java,” in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 815–816.
- [17] R. Patton, *Software Testing*, 2nd ed. Paul Boger, 7 2005.
- [18] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, “Jtexpert at the third unit testing tool competition,” in *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*, 2015, pp. 52–55.
- [19] M. Schafer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *IEEE Transactions on Software Engineering*, vol. 50, pp. 85–105, 1 2024.
- [20] M. D. M. Suffian, L. F. Ann, M. Subakin, S. S. Tzuaan, and F. F. M. Nazri, “Consistency of system test case results for software capability rating,” in *Consistency of System Test Case Results for Software Capability Rating*, 2018.
- [21] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.
- [22] B. Yetistiren, I. Ozsoy, and E. Tuzun, “Assessing the quality of github copilot’s code generation,” in *Assessing the quality of GitHub copilot’s code generation*. Association for Computing Machinery, Inc, 11 2022, pp. 62–71.
- [23] M. L. Zhiqiang Yuan, Yiling Lou, S. Ding, K. Wang, Y. Chen, and X. Peng, “No more manual tests? evaluating and improving chatgpt for unit test generation,” *arXiv*, May 2023.

A.1 RQ1: Sorted Relevance Score from Scenario One.

Group Name	Function ID	HumanEval ID	Number Relevant Test Cases from Codiumate	Total Number of Test Cases Suggested by Codiumate	Relevance Score (%)	Number of Test Cases with Duplicate Name
Group 1	14	76	2	19	10.53	0
	9	44	2	15	13.33	0
	13	67	2	15	13.33	0
	23	106	3	20	15.00	0
	5	11	3	15	20.00	0
	7	23	3	15	20.00	0
	11	58	3	14	21.43	0
Group 2	21	96	3	14	21.43	0
	2	3	5	19	26.32	0
	4	8	6	19	31.58	0
	1	2	7	20	35.00	0
	6	13	5	14	35.71	0
	8	28	5	14	35.71	0
	10	51	5	14	35.71	0
	28	148	5	13	38.46	1
Group 3	12	64	6	15	40.00	2
	15	77	7	15	46.67	0
	3	7	8	15	53.33	0
	24	119	8	15	53.33	2
	29	150	8	15	53.33	3
	22	99	10	18	55.56	3
	26	138	9	13	69.23	2
Group 4	20	93	13	18	72.22	1
	17	79	11	14	78.57	7
	25	125	16	20	80.00	3
	30	152	16	20	80.00	1
	18	81	18	20	90.00	0
	27	145	14	15	93.33	4
	16	78	20	20	100.00	0
Total	19	82	14	14	100.00	0
	-	-	237	487	-	29
Total without Duplicate Test Cases			208	458	-	
Average			-	-	47.97	

Figure A.1: Function groups sorted by Relevance Scores

A.2 RQ1.2 Relevance Scores From Scenario One and Scenario Two.

Scenario 1: Groups	Function ID	HumanEval ID	RC1 (%)	RC2 (%)
Group 1	14	76	10.53	71.43
	9	44	13.33	42.86
	13	67	13.33	60.00
	23	106	15.00	50.00
	5	11	20.00	42.86
	7	23	20.00	100.00
	11	58	21.43	42.86
	21	96	21.43	57.14
Group 2	2	3	26.32	57.14
	4	8	31.58	71.43
	1	2	35.00	100.00
	6	13	35.71	50.00
	8	28	35.71	71.43
	10	51	35.71	80.00
	28	148	38.46	55.56
	12	64	40.00	54.55
	15	77	46.67	85.71
	Group 3	3	7	53.33
24		119	53.33	71.43
29		150	53.33	100.00
22		99	55.56	77.78
26		138	69.23	87.50
20		93	72.22	88.89
Group 4	17	79	78.57	83.33
	25	125	80.00	62.50
	30	152	80.00	85.71
	18	81	90.00	76.47
	27	145	93.33	66.67
	16	78	100.00	80.00
<u>Average</u>			<u>47.97</u>	<u>70.44</u>

Figure A.2: Relevance Scores from Scenario One and Scenario Two.

A.3 RQ2: False Positive Distribution Among Groups

Group Name	Function ID	HumanEval ID	Relevant Test Cases from RQ 1	False Positives	False Positive Rate (FPR)	Number of Test Cases with Duplicate Name	Number of Failed Test Cases	Number of Past Test Cases
Group 1	14	76	2	0	0	0	0	2
0% <= RC <25%	9	44	2	0	0	0	1	1
	13	67	2	0	0	0	0	2
	23	106	3	0	0	0	0	3
	5	11	3	0	0	0	1	2
	7	23	3	0	0	0	0	3
	11	58	3	0	0	0	0	3
	21	96	3	0	0	0	0	3
Group 2	2	3	5	1	20	0	1	4
25% < RC <= 50%	4	8	6	1	16.67	0	1	5
	1	2	7	1	14.29	0	2	5
	6	13	5	0	0	0	2	3
	8	28	5	1	20	0	1	4
	10	51	5	0	0	0	0	5
	28	148	5	1	20	1	1	3
	12	64	6	1	16.67	2	2	2
	15	77	7	1	14.29	0	1	6
Group 3	3	7	8	0	0	0	0	8
50% < RC <= 75%	24	119	8	0	0	2	0	6
	29	150	8	2	25	3	2	3
	22	99	10	0	0	3	0	7
	26	138	9	0	0	2	0	7
	20	93	13	9	69.23	1	9	3
Group 4	17	79	11	1	9.09	7	1	3
75% < RC <= 100%	25	125	16	0	0	3	3	10
	30	152	16	1	6.25	1	1	14
	18	81	18	8	44.44	0	8	10
	27	145	14	1	7.14	4	3	7
	16	78	20	0	0	0	0	20
	19	82	14	2	14.29	0	2	12
Total	-	-	237	31	-	29	42	166
Average	-	-	7.90	1.03	9.91	0.97	1.40	5.53
-	-	-	-	-	-	-	-	-
-	-	-	False Positive Percentage of Total Test cases	14.90%	-	-	-	-
-	-	-	False Positive Percentage of Failed Test Cases	73.81%	-	-	-	-

Figure A.3: False Positive result from scenario one.

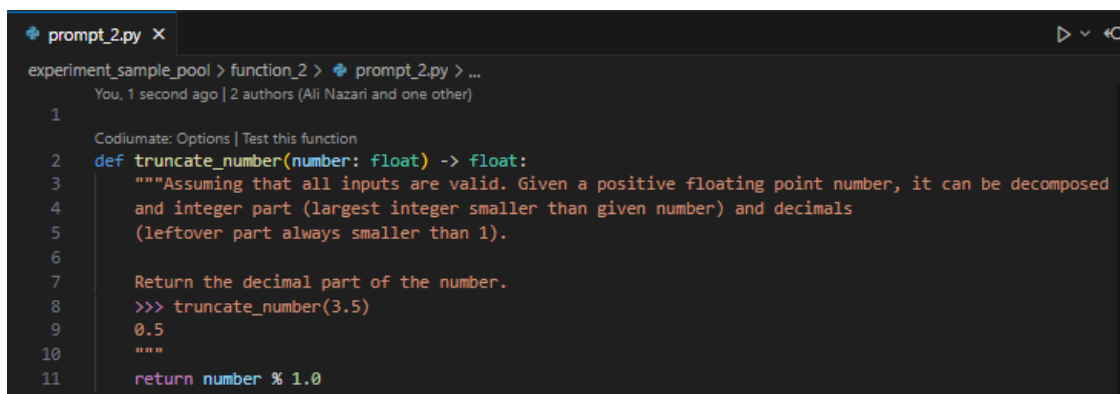
Group Name	Function ID	HumanEval ID	Number of change	Total Number of Executions	Result Consistency (%)
Group 1	14	76	0	5	100
	9	44	0	5	100
	13	67	0	5	100
	23	106	0	5	100
	5	11	0	5	100
	7	23	0	5	100
	11	58	0	5	100
	21	96	0	5	100
Group 2	2	3	0	5	100
	4	8	0	5	100
	1	2	0	5	100
	6	13	0	5	100
	8	28	0	5	100
	10	51	0	5	100
	28	148	0	5	100
	12	64	0	5	100
	15	77	0	5	100
Group 3	3	7	0	5	100
	24	119	0	5	100
	29	150	0	5	100
	22	99	0	5	100
	26	138	0	5	100
	20	93	0	5	100
Group 4	17	79	0	5	100
	25	125	0	5	100
	30	152	0	5	100
	18	81	0	5	100
	27	145	0	5	100
	16	78	0	5	100
	19	82	0	5	100
Total	-	-	0	150	-
Average Result Consistency	-	-	-	-	100

Figure A.4: Result consistency from test runs.

Appendix B

Sample Functions, Selection Criteria, and Relevant Test Cases

B.1 Sample 1



```
prompt_2.py x
experiment_sample_pool > function_2 > prompt_2.py > ...
You, 1 second ago | 2 authors (Ali Nazari and one other)

1
Codiumate: Options | Test this function
2 def truncate_number(number: float) -> float:
3     """Assuming that all inputs are valid. Given a positive floating point number, it can be decomposed
4     and integer part (largest integer smaller than given number) and decimals
5     (leftover part always smaller than 1).
6
7     Return the decimal part of the number.
8     >>> truncate_number(3.5)
9     0.5
10    """
11    return number % 1.0
```

Figure B.1: Sample 1: Function Under Test

```

1  # Function 2 Test Case Criteria {#1}
2
3  ## Test Cases for `truncate_number` Function
4
5  ### Boundary Values:
6  Consider test cases where the input number is at the boundary between two integers.
7  Since the function returns the decimal part of the number, testing with values like 3.5,
8  where the decimal part is exactly 0.5, would be relevant.
9
10 | Test Case | Input           | Expected Output |
11 |-----|-----|-----|
12 | 1         | `3.5`           | `0.5`           |
13
14 ### Typical Values:
15 Test with typical positive floating-point numbers that may occur in practical scenarios.
16 These could include numbers with one or more decimal places such as 2.3, 10.56, or 123.789.
17
18 | Test Case | Input           | Expected Output |
19 |-----|-----|-----|
20 | 1         | `2.3`           | `0.3`           |
21 | 2         | `10.56`         | `0.56`          |
22 | 3         | `123.789`       | `0.789`         |
23
24 ### Special Cases:
25 Consider testing with special cases mentioned or implied in the docstring. For instance:
26 - Test with zero as input (0.0) to ensure the function handles it properly.
27 - Test with very small or very large positive floating-point numbers to assess the
28 function's behavior with extreme values.
29
30 | Test Case | Input           | Expected Output |
31 |-----|-----|-----|
32 | 4         | `0.0`           | `0.0`           |
33 | 5         | `1233434232332.0323890432` | `0.0323890432` |
34
35 ### Accuracy Testing:
36 Include test cases to verify the accuracy and precision of the function's output,
37 especially concerning floating-point arithmetic. This could involve testing with
38 numbers very close to whole numbers to check for any rounding behavior.
39
40 | Test Case | Input           | Expected Output |
41 |-----|-----|-----|
42 | 6         | `10.000001`     | `0.000001`     |
43 | 6         | `10.999999`     | `0.999999`     |

```

Figure B.2: Sample 1: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_2.prompt_2 import truncate_number
5 # Generated by CodiumAI
6
7 import pytest
8
9 Codiumate: Add more tests | You, 1 minute ago | 2 authors (Ali Nazari and one other)
10 class TestTruncateNumber:
11     # Matche test cases 1,2,3 Returns the decimal part of a positive floating point number.
12     Codiumate: Add more tests
13     def test_decimal_part_positive_number(self):
14         assert truncate_number(3.5) == 0.5
15         assert truncate_number(2.75) == 0.75
16         assert truncate_number(10.123456789) == 0.123456789
17
18     # Matche test cases 6 Returns the expected decimal part when the input is the smallest positive floating po
19     Codiumate: Add more tests
20     def test_decimal_part_smallest_positive_number(self):
21         assert truncate_number(1e-323) == 1e-323
22
23     # Matche test cases 4 Returns 0.0 when the input is 0.0.
24     Codiumate: Add more tests
25     def test_returns_zero_when_input_is_zero(self):
26         assert truncate_number(0.0) == 0.0
27
28     # Returns the expected decimal part when the input is the largest positive floating point number.
29     Codiumate: Add more tests
30     def test_largest_positive_float(self):
31         assert truncate_number(1.7976931348623157e+308) == 0.0
32
33     # Returns the expected decimal part when the input is the smallest subnormal positive floating point number
34     Codiumate: Add more tests
35     def test_smallest_subnormal_positive_number(self):
36         assert truncate_number(2.2250738585072014e-308) == 2.2250738585072014e-308
37
38     # Returns the expected decimal part when the input is the largest subnormal positive floating point number.
39     Codiumate: Add more tests
40     def test_largest_subnormal_positive_number(self):
41         assert truncate_number(2.2250738585072014e-308) == 2.2250738585072014e-308
42
43     # Matche test cases 5 Returns the expected decimal part when the input is a floating point number with a la
44     Codiumate: Add more tests
45     def test_decimal_part_large_number_of_decimal_places(self):
46         assert truncate_number(3.141592653589793238462643383279502884197169399375105820974944592307816406286208
47         assert truncate_number(2.718281828459045235360287471352662497757247093699959574966967627724076630353547
48         assert truncate_number(1.618033988749894848204586834365638117720309179805762862135448622705260462818902

```

Figure B.3: Sample 1: Relevant Test Cases

B.2 Sample 2

```

Codiumate: Options | Test this function
2 def truncate_number(number: float) -> float:
3     """Assuming that all inputs are valid. Given a positive floating point number,
4     it can be decomposed into an integer part (largest integer smaller than given number)
5     and decimals (leftover part always smaller than 1).
6
7     Return the decimal part of the number.
8     >>> truncate_number(3.5)
9     0.5
10    """
11    return number % 1.0

```

Figure B.4: Sample 2

```

1 # Function 3 Test Case Criteria {#2} You, 2 months ago • Added all README<function-name> ...
2
3 ## Test Cases for `below_zero` Function
4
5 ### Boundary Values:
6 Test with boundary values where the balance may potentially go below zero. This includes scenarios where:
7 - The operations list is empty.
8 - The first operation results in a negative balance.
9 - The balance remains positive throughout the operations.
10
11 | Test Case | Input          | Expected Output |
12 |-----|-----|-----|
13 | 1         | `[]`          | `False`         |
14 | 2         | `[0]`         | `False`         |
15 | 3         | `[1, 2, 3]`   | `False`         |
16 | 4         | `[0, -5, 10]` | `True`          |
17 | 5         | `[-5, -10]`  | `True`          |
18
19 ### Typical Values:
20 Test with typical deposit and withdrawal operations that could occur in a real-world scenario. These could include:
21 - Various combinations of positive and negative integers.
22 - Cases where the balance becomes negative at some point.
23
24 | Test Case | Input          | Expected Output |
25 |-----|-----|-----|
26 | 5         | `[1, 2, -4, 5]` | `True`          |
27 | 5         | `[100, -50, 25]` | `False`         |
28 | 5         | `[10, -15, 5]`  | `True`          |
29
30 ### Special Cases:
31 Consider special cases mentioned or implied in the docstring. For instance:
32 - Test with a single operation causing a negative balance.
33 - Test with a large number of operations.
34
35 | Test Case | Input          | Expected Output |
36 |-----|-----|-----|
37 | 6         | `[-10]`       | `True`          |
38 | 7         | `[10]`        | `False`         |

```

Figure B.5: Sample 2: Selection Criteria

```

codiumate_result > s2_f3.py > TestBelowZero
1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  from experiment_sample_pool.function_3.prompt_3 import below_zero
5  # Generated by CodiumAI
6
7
8  import pytest
9
10 Codiumate: Add more tests
11 class TestBelowZero:
12     # Matches with test case 1
13     # The function returns False when given an empty list.
14     Codiumate: Add more tests
15     def test_empty_list_returns_false(self):
16         assert below_zero([]) == False
17
18     # Matches with test case 3
19     # The function returns False when given a list with only positive integers.
20     Codiumate: Add more tests
21     def test_returns_false_with_positive_integers(self):
22         assert below_zero([1, 2, 3]) == False
23
24     # Matches with test case 6
25     # The function returns False when given a list with only one negative integer.
26     Codiumate: Add more tests
27     def test_returns_false_with_one_negative_integer(self):
28         assert below_zero([-5]) == False
29
30     # Matches with test case 5
31     # The function returns True when given a list with a mix of positive and negative integers,
32     # and the balance falls below zero.
33     Codiumate: Add more tests
34     def test_balance_falls_below_zero(self):
35         assert below_zero([1, 2, -4, 5]) == True
36
37     # Matches with test case 3
38     # The function returns False when given a list with a mix of positive and negative integers,
39     # but the balance never falls below zero.
40     Codiumate: Add more tests
41     def test_balance_never_falls_below_zero(self):
42         assert below_zero([1, 2, 3]) == False

```

Figure B.6: Sample 2: Relevant Test Cases

B.3 Sample 3

```

1  from typing import List          You, 2 months ago * added 15 selection criteria
2
3
4  Codiumate: Options | Test this function
5  def filter_by_substring(strings: List[str], substring: str) -> List[str]:
6      """Assuming that all inputs are valid. Filter an input list of strings only for ones that contain given substring
7      >>> filter_by_substring([], 'a')
8          []
9      >>> filter_by_substring(['abc', 'bacd', 'cde', 'array'], 'a')
10         ['abc', 'bacd', 'array']
11      """
12     return [string for string in strings if substring in string]

```

Figure B.7: Sample 3: Function Under Test

```

1  ✓ # Test Case Selection Criteria for filter_by_substring Function          You, 2 months ago * added 15 selection cri
2
3  ✓ ## Test Cases for `filter_by_substring` Function
4
5  ✓ ### Boundary Values:
6  Test with boundary values where `strings` is empty or `substring` is an empty string.
7
8  ✓ | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1 | `[], 'a' | `[]` |
11 | 2 | `['abc', 'bacd', 'cde'], ''` | `[]` |
12
13 ✓ ### Typical Values:
14 Test with typical inputs where `strings` contains various strings and `substring` appears in some of them.
15
16 ✓ | Test Case | Input | Expected Output |
17 |-----|-----|-----|
18 | 3 | `['abc', 'bacd', 'cde'], 'a'` | `['abc', 'bacd']` |
19 | 3 | `['hello', 'world'], 'o'` | `['hello', 'world']` |
20 | 4 | `['apple', 'banana', 'cherry'], 'na'` | `['banana']` |
21 | 4 | `['grunt', 'trumpet', 'prune', 'gruesome'], 'run'` | `['grunt', 'prune']` |
22 | 4 | `['xxx', 'asd', 'xy', 'john doe', 'xxxAAA', 'xxx'], 'xxx'` | `['xxx', 'xxxAAA', 'xxx']` |
23 | 5 | `['abc', 'def', 'ghi'], 'j'` | `[]` |
24 | 5 | `['cat', 'dog', 'fish'], 'z'` | `[]` |
25
26 ✓ ### Special Cases:
27 Consider special cases mentioned or implied in the docstring. For instance:
28 - Test with `strings` containing one string.
29 - Test with `strings` containing one long string.
30
31 ✓ | Test Case | Input | Expected Output |
32 |-----|-----|-----|
33 | 6 | `['hello'], 'llo'` | `['hello']` |
34 | 7 | `['hello', 'world'], 'longlongword'` | `[]` |
35

```

Figure B.8: Sample 3: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  from experiment_sample_pool.function_7.prompt_7 import filter_by_substring
5
6
7  # Generated by CodiumAI
8
9  import pytest
10
11  Codiumate: Add more tests
12  class TestFilterBySubstring:
13      # Matches with test case 3
14      # Test with a list of strings containing the substring
15      Codiumate: Add more tests
16      def test_list_with_substring(self):
17          strings = ['abc', 'bacd', 'cde', 'array']
18          substring = 'a'
19          expected_result = ['abc', 'bacd', 'array']
20
21          assert filter_by_substring(strings, substring) == expected_result
22
23      # Matches with test case 5
24      # Test with a list of strings not containing the substring
25      Codiumate: Add more tests
26      def test_list_without_substring(self):
27          strings = ['abc', 'bacd', 'cde', 'array']
28          substring = 'x'
29          expected_result = []
30
31          assert filter_by_substring(strings, substring) == expected_result
32
33      # Matches with test case 3 (duplicate test case: test_list_with_substring)
34      # Test with multiple substrings in the same string
35      Codiumate: Add more tests
36      def test_multiple_substrings(self):
37          strings = ['abc', 'bacd', 'cde', 'array']
38          substring = 'a'
39          expected_result = ['abc', 'bacd', 'array']
40
41          assert filter_by_substring(strings, substring) == expected_result
42
43      # Matches with test case 1
44      # Test with an empty list of strings
45      Codiumate: Add more tests
46      def test_empty_list(self):
47          strings = []
48          substring = 'a'
49          expected_result = []
50
51          assert filter_by_substring(strings, substring) == expected_result
52
53      # Matches with test case 2
54      # Test with an empty substring
55      Codiumate: Add more tests
56      def test_empty_substring(self):
57          strings = ['abc', 'bacd', 'cde', 'array']
58          substring = ''
59          expected_result = ['abc', 'bacd', 'cde', 'array']

```

Figure B.9: Sample 3: Relevant Test Cases 1/2

```
55
56     assert filter_by_substring(strings, substring) == expected_result
57
58     # Test with a list of strings where all strings are empty
59     Codiumate: Add more tests
60     def test_list_with_empty_strings(self):
61         strings = ['', '', '', '']
62         substring = 'a'
63         expected_result = []
64
65         assert filter_by_substring(strings, substring) == expected_result
66
67     # Matches with test case 7
68     # Test with a very long substring
69     Codiumate: Add more tests
70     def test_long_substring(self):
71         strings = ['abc', 'bacd', 'cde', 'array']
72         substring = 'verylongsubstring'
73         expected_result = []
74
75         assert filter_by_substring(strings, substring) == expected_result
76
77     # Test with a substring that contains special characters
78     Codiumate: Add more tests
79     def test_substring_with_special_characters(self):
80         strings = ['abc', 'bacd', 'cde', 'array']
81         substring = '!@#$$'
82         expected_result = []
83
84         assert filter_by_substring(strings, substring) == expected_result
```

Figure B.10: Sample 3: Relevant Test Cases 2/2

B.4 Sample 4

```

2 from typing import List, Tuple
3
4
5 Codumate: Options | Test this function
6 def sum_product(numbers: List[int]) -> Tuple[int, int]:
7     """Assuming that all inputs are valid. For a given list of integers, return a tuple consisting of a sum and a product of all the integers in a list.
8     Empty sum should be equal to 0 and empty product should be equal to 1.
9     >>> sum_product([])
10        (0, 1)
11     >>> sum_product([1, 2, 3, 4])
12        (10, 24)
13     """
14     sum_value = 0
15     prod_value = 1
16
17     for n in numbers:
18         sum_value += n
19         prod_value *= n
20     return sum_value, prod_value

```

Figure B.11: Sample 4: Function Under Test

```

1 # Function 8 Test Case Criteria (#3) You, 2 months ago · Added all README(function-name)
2
3 ## Test Cases for `sum_product` Function
4
5 ### Boundary Values:
6 Test with boundary values where the list of integers is empty and contains only one element. These scenarios should cover the cases where:
7 - The input list is empty.
8 - The input list contains a single element.
9
10 | Test Case | Input | Expected Output |
11 |-----|-----|-----|
12 | 1 | `[ ]` | `(0, 1)` |
13 | 2 | `[5]` | `(5, 5)` |
14
15 ### Typical Values:
16 Test with typical lists of integers that may occur in a real-world scenario. These could include:
17 - Various combinations of positive and negative integers.
18 - Cases with multiple elements in the list.
19
20 | Test Case | Input | Expected Output |
21 |-----|-----|-----|
22 | 3 | `[1, 2, 3, 4]` | `(10, 24)` |
23 | 3 | `[5, 5, 5, 5, 5]` | `(25, 3125)` |
24 | 4 | `[-1, 2, -3, 4]` | `(2, 24)` |
25
26 ### Special Cases:
27 Consider special cases mentioned or implied in the docstring. For instance:
28 - Test with a large number of elements in the list.
29
30 | Test Case | Input | Expected Output |
31 |-----|-----|-----|
32 | 5 | `[1, 1000]` | `(1001, 1000)` |
33 | 5 | `[-1, 1000]` | `(999, -1000)` |
34 | 5 | `[1, -1, 500]` | `(500, -500)` |
35 | 5 | `[1000, 1000, -100]` | `(2900, -100000000)` |
36
37 ### Regression Testing:
38 Include test cases that cover any bugs or issues identified in the past to ensure they have been addressed and do not reoccur.
39
40 | Test Case | Input | Expected Output |
41 |-----|-----|-----|
42 | 6 | `[0, 0, 0, 0, 0]` | `(0, 0)` |
43 | 7 | `[1, 1, 1, 1, 1, 0]` | `(5, 0)` |
44

```

Figure B.12: Sample 4: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_8.prompt_8 import sum_product
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestSumProduct:
13     # Matches with test case 3
14     # Returns a tuple with sum and product of a list of integers
15     Codiumate: Add more tests
16     def test_sum_product_with_positive_integers(self):
17         result = sum_product([1, 2, 3, 4])
18         assert result == (10, 24)
19
20     # Matches with test case 1
21     # Returns (0, 1) when an empty list is passed as argument
22     Codiumate: Add more tests
23     def test_empty_list(self):
24         result = sum_product([])
25         assert result == (0, 1)
26
27     # Matches with test case 2
28     # Works correctly with a list of length 1
29     Codiumate: Add more tests
30     def test_list_length_1(self):
31         result = sum_product([5])
32         assert result == (5, 5)
33
34     # Works correctly with a list containing only negative integers
35     Codiumate: Add more tests
36     def test_behaviour_with_negative_integers(self):
37         result = sum_product([-1, -2, -3, -4])
38         assert result == (-10, -24)
39
40     # Matches with test case 4
41     # Works correctly with a list containing both positive and negative integers
42     Codiumate: Add more tests
43     def test_sum_product_with_positive_and_negative_integers(self):
44         result = sum_product([-1, 2, -3, 4])
45         assert result == (2, 24)
46
47     # Matches with test case 6
48     # Works correctly with a list containing only zeros
49     Codiumate: Add more tests
50     def test_correct_with_zeros(self):
51         result = sum_product([0, 0, 0, 0])
52         assert result == (0, 0)

```

Figure B.13: Sample 4: Relevant Test Cases

B.5 Sample 5

```

1  from typing import List      Ali Nazari, 3 months ago • add experiment code and canonical tests
2
3
4  Codiumate: Options | Test this function
5  def string_xor(a: str, b: str) -> str:
6      """Assuming that all inputs are valid. Input are two strings a and b consisting only of 1s and 0s.
7      Perform binary XOR on these inputs and return result also as a string.
8      >>> string_xor('010', '110')
9      '100'
10     """
11     def xor(i, j):
12         if i == j:
13             return '0'
14         else:
15             return '1'
16     return ''.join(xor(x, y) for x, y in zip(a, b))
17

```

Figure B.14: Sample 5: Function Under Test

```

1  # Function 11 Test Case Criteria {#5}
2
3  ## Test Cases for `string_xor` Function
4
5  ### Boundary Values:
6  Test with boundary values where the strings are empty and contain only one character. These scenarios should cover the cases where:
7  - The input strings are empty.
8  - The input strings contain a single character.
9
10 | Test Case | Input          | Expected Output |
11 |-----|-----|-----|
12 | 1         | `('', '')`    | ``              |
13 | 2         | `('0', '0')` | `0`             |
14 | 3         | `('1', '1')` | `0`             |
15 | 4         | `('1', '0')` | `1`             |
16 | 5         | `('0', '1')` | `1`             |
17
18 ### Typical Values:
19 Test with typical strings of binary numbers that may occur in a real-world scenario. These could include:
20 - Various combinations of binary strings with different lengths.
21 - Cases with multiple elements in the strings.
22
23 | Test Case | Input          | Expected Output |
24 |-----|-----|-----|
25 | 6         | `('010', '110')` | `100`          |
26 | 6         | `('1010', '1110')` | `0100`         |
27 | 6         | `('111', '000')` | `111`          |
28
29 ### Special Cases:
30 Consider special cases mentioned or implied in the docstring. For instance:
31 - Test with a large number of characters in the strings.
32
33 | Test Case | Input          | Expected Output |
34 |-----|-----|-----|
35 | 7         | `('11111111', '00000000')` | `11111111`    |
36 | 7         | `('00001111', '11111111')` | `11110000`    |
37
38 ### Regression Testing:
39 Include test cases that cover any bugs or issues identified in the past to ensure they have been addressed and do not reoccur.
40
41 | Test Case | Input          | Expected Output |
42 |-----|-----|-----|
43 | 6         | `('0101', '1010')` | `1111`         |
44 | 6         | `('0000', '0000')` | `0000`         |
45

```

Figure B.15: Sample 5: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_11.prompt_11 import string_xor
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestStringXor:
13     # Matches with test case 6, 7
14     # The function returns the correct XOR of two binary strings.
15     Codiumate: Add more tests
16     def test_correct_xor(self):
17         assert string_xor('010', '110') == '100'
18         assert string_xor('101010', '010101') == '111111'
19         assert string_xor('1111', '0000') == '1111'
20         assert string_xor('11111111', '00000000') == '11111111'
21
22     # Matches with test case 2, 3, 4, 5
23     # The function can handle binary strings of length 1.
24     Codiumate: Add more tests
25     def test_binary_string_of_length_1(self):
26         assert string_xor('0', '0') == '0'
27         assert string_xor('0', '1') == '1'
28         assert string_xor('1', '0') == '1'
29         assert string_xor('1', '1') == '0'
30
31     # Matches with test case 1, 6
32     # The function can handle binary strings of different lengths.
33     Codiumate: Add more tests
34     def test_binary_string_lengths(self):
35         assert string_xor('010', '110') == '100'
36         assert string_xor('1010', '110') == '1110'
37         assert string_xor('101010', '110') == '111100'
38         assert string_xor('101010', '110110') == '011100'
39         assert string_xor('101010', '') == ''
40         assert string_xor('', '110110') == ''

```

Figure B.16: Sample 5: Relevant Test Cases

B.6 Sample 6

```

2
Codiumate: Options | Test this function
3 def greatest_common_divisor(a: int, b: int) -> int:
4     """Assuming that all inputs are valid. Return a greatest common divisor of two integers a and b
5     >>> greatest_common_divisor(3, 5)
6         1
7     >>> greatest_common_divisor(25, 15)
8         5
9     """
10    while b:
11        a, b = b, a % b
12    return a

```

Figure B.17: Sample 6: Function Under Test

```

1 # Function 13 Test Case Criteria {#6}
2
3 ## Test Cases for `greatest_common_divisor` Function
4
5 ### Boundary Values:
6 Test with boundary values where the inputs are at their minimum or maximum allowable values.
7 This includes scenarios where:
8 - One or both inputs are zero.
9 - One input is a very large positive or negative number.
10 - Both inputs are the same number.
11
12 | Test Case | Input          | Expected Output |
13 |-----|-----|-----|
14 | 1         | `(0, 5)`      | `5`             |
15 | 2         | `(25, 0)`     | `25`            |
16 | 3         | `(1000000, 5)` | `5`             |
17 | 4         | `(0, 0)`      | `0`             |
18 | 5         | `(-10, 5)`   | `5`             |
19 | 6         | `(10, 10)`   | `10`            |
20
21 ### Typical Values:
22 Test with typical pairs of integers that may occur in a real-world scenario.
23 These could include pairs of relatively prime numbers,
24 pairs with a common divisor, and pairs with a large common divisor.
25
26 | Test Case | Input          | Expected Output |
27 |-----|-----|-----|
28 | 7         | `(3, 5)`      | `1`             |
29 | 7         | `(25, 15)`   | `5`             |
30 | 7         | `(56, 72)`   | `8`             |
31 | 7         | `(90, 15)`   | `15`            |
32 | 7         | `(168, 36)`  | `12`            |
33
34 ### Special Cases:
35 Consider special cases mentioned or implied in the docstring. For instance:
36 - Test with negative integers.
37
38 | Test Case | Input          | Expected Output |
39 |-----|-----|-----|
40 | 8         | `(-12, -18)` | `6`             |
41

```

Figure B.18: Sample 6: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_13.prompt_13 import greatest_common_divisor
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestGreatestCommonDivisor:
13     # Matches with test case 7
14     # The function returns the correct greatest common divisor for two positive integers.
15     Codiumate: Add more tests
16     def test_greatest_common_divisor_positive_integers(self):
17         assert greatest_common_divisor(3, 5) == 1
18         assert greatest_common_divisor(25, 15) == 5
19
20     # Matches with test case 8
21     # The function returns the correct greatest common divisor for two negative integers.
22     Codiumate: Add more tests
23     def test_greatest_common_divisor_negative_integers(self):
24         assert greatest_common_divisor(-3, -5) == 1
25         assert greatest_common_divisor(-25, -15) == 5
26
27     # Matches with test case 8
28     # The function returns the correct greatest common divisor for one positive and one negative integer.
29     Codiumate: Add more tests
30     def test_greatest_common_divisor_positive_and_negative_integers(self):
31         assert greatest_common_divisor(3, -5) == 1
32         assert greatest_common_divisor(-25, 15) == 5
33
34     # Matches with test case 1, 2
35     # The function returns the correct greatest common divisor for when one of the integers is 0.
36     Codiumate: Add more tests
37     def test_greatest_common_divisor_with_one_integer_zero(self):
38         assert greatest_common_divisor(0, 5) == 5
39         assert greatest_common_divisor(25, 0) == 25
40
41     # Matches with test case 4
42     # The function returns the correct greatest common divisor for when both integers are 0.
43     Codiumate: Add more tests
44     def test_greatest_common_divisor_zero_integers(self):
45         assert greatest_common_divisor(0, 0) == 0
```

Figure B.19: Sample 6: Relevant Test Cases

B.7 Sample 7

```

Codiumate: Options | Test this function
3 def strlen(string: str) -> int:
4     """Assuming that all inputs are valid.
5     Return length of given string
6     >>> strlen('')
7     0
8     >>> strlen('abc')
9     3
10    """
11    return len(string)
12

```

Figure B.20: Sample 7: Function Under Test

```

1 # Function 23 Test Case Criteria {#7}
2
3 ## Test Cases for `strlen` Function
4
5 ### Boundary Values:
6 Test with boundary values where the input string is empty or contains only one character.
7 These scenarios should cover the cases where:
8 - The input string is empty.
9 - The input string contains a single character.
10
11 | Test Case | Input          | Expected Output |
12 |-----|-----|-----|
13 | 1         | ``             | `0`             |
14 | 2         | `a`           | `1`             |
15 | 3         | ` `           | `1`             |
16
17 ### Typical Values:
18 Test with typical strings of varying lengths that may occur in a real-world scenario.
19 These could include strings with different lengths and containing various characters.
20
21 | Test Case | Input          | Expected Output |
22 |-----|-----|-----|
23 | 4         | `abc`         | `3`             |
24 | 4         | `hello`       | `5`             |
25 | 4         | `1234567890` | `10`            |
26
27 ### Special Cases:
28 Consider special cases mentioned or implied in the docstring. For instance:
29 - Test with Unicode characters.
30 - Test with whitespace characters.
31
32 | Test Case | Input          | Expected Output |
33 |-----|-----|-----|
34 | 5         | `😄😄😄`         | `3`             |
35 | 6         | `a b c`       | `5`             |
36 | 7         | `a\b\n\c`    | `5`             |
37

```

Figure B.21: Sample 7: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_23.prompt_23 import strlen
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestStrlen:
13     # Matches test case 4
14     # Returns the length of a non-empty string
15     Codiumate: Add more tests
16     def test_returns_length_of_non_empty_string(self):
17         assert strlen('abc') == 3
18
19     # Matches test case 1
20     # Returns 0 for an empty string
21     Codiumate: Add more tests
22     def test_empty_string(self):
23         assert strlen('') == 0
24
25     # Matches test case 7
26     # Handles strings with multiple lines
27     Codiumate: Add more tests
28     def test_handles_strings_with_multiple_lines(self):
29         assert strlen('abc\n123\nxyz') == 11
```

Figure B.22: Sample 7: Relevant Test Cases

B.8 Sample 8

```

1  from typing import List
2
3
4  Codiumate: Options | Test this function
5  def concatenate(strings: List[str]) -> str:
6      """Assuming that all inputs are valid.
7      Concatenate list of strings into a single string
8      >>> concatenate([])
9      ''
10     >>> concatenate(['a', 'b', 'c'])
11     'abc'
12     """
13     return ''.join(strings)

```

Figure B.23: Sample 8: Function Under Test

```

1  # Function 28 Test Case Criteria {#8}
2
3  ## Test Cases for `concatenate` Function
4
5  ### Boundary Values:
6  Test with boundary values where the input list is empty or contains only one string.
7  These scenarios should cover the cases where:
8  - The input list is empty.
9  - The input list contains a single string.
10
11 | Test Case | Input          | Expected Output |
12 |-----|-----|-----|
13 | 1         | `[]`           | `''`            |
14 | 2         | `['a']`        | `'a'`           |
15 | 3         | `['']`         | `''`            |
16
17 ### Typical Values:
18 Test with typical lists of strings that may occur in a real-world scenario.
19 These could include lists with different lengths and containing various strings.
20
21 | Test Case | Input          | Expected Output |
22 |-----|-----|-----|
23 | 4         | `['a', 'b', 'c']` | `abc`           |
24 | 4         | `['hello', 'world']` | `helloworld`   |
25 | 4         | `['123', '456', '789']` | `123456789`    |
26
27 ### Special Cases:
28 Consider special cases mentioned or implied in the docstring. For instance:
29 - Test with strings containing special characters.
30 - Test with strings containing whitespace characters.
31
32 | Test Case | Input          | Expected Output |
33 |-----|-----|-----|
34 | 5         | `['a ', ' b', ' c']` | `a b c`         |
35

```

Figure B.24: Sample 8: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_28.prompt_28 import concatenate
5
6 # Generated by CodiumAI
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestConcatenate:
12     # Matches with test case 1
13     # The function correctly concatenates a list of strings with no elements.
14     Codiumate: Add more tests
15     def test_concatenate_empty_list(self):
16         strings = []
17         result = concatenate(strings)
18         assert result == ''
19
20     # Matches with test case 2
21     # The function correctly concatenates a list of strings with one element.
22     Codiumate: Add more tests
23     def test_concatenate_single_element(self):
24         strings = ['abc']
25         result = concatenate(strings)
26         assert result == 'abc'
27
28     # Matches with test case 4
29     # The function correctly concatenates a list of strings with multiple elements.
30     Codiumate: Add more tests
31     def test_concatenate_multiple_elements(self):
32         strings = ['a', 'b', 'c']
33         result = concatenate(strings)
34         assert result == 'abc'
35
36     # The function should handle special characters and escape sequences correctly.
37     Codiumate: Add more tests
38     def test_concatenate_special_characters(self):
39         strings = ['a', 'b', '\n', '\t', '\\']
40         result = concatenate(strings)
41         assert result == 'ab\n\t\\'
42
43     # Mathces with test case 5
44     # The function should handle leading and trailing whitespaces in the input strings correctly.
45     Codiumate: Add more tests
46     def test_handle_leading_and_trailing_whitespaces(self):
47         strings = [' a', 'b ', ' c ']
48         result = concatenate(strings)
49         assert result == ' ab c '

```

Figure B.25: Sample 8: Relevant Test Cases

B.9 Sample 9

```
Codiumate: Options | Test this function
3 def change_base(x: int, base: int):
4     """Assuming that all inputs are valid. This function changes numerical base of input number x to base.
5     return string representation after the conversion.
6     base numbers are less than 10.
7     >>> change_base(8, 3)
8     '22'
9     >>> change_base(8, 2)
10    '1000'
11    >>> change_base(7, 2)
12    '111'
13    """
14    ret = ""
15    while x > 0:
16        ret = str(x % base) + ret
17        x //= base
18    return ret
19
```

Figure B.26: Sample 9: Function Under Test

```

1  # Function 44 Test Case Criteria {#9}
2
3  ## Test Cases for `change_base` Function
4
5  ** Example x = 8, base = 2 **
6      8 divided by 2 equals 4 with a remainder of 0.
7      4 divided by 2 equals 2 with a remainder of 0.
8      2 divided by 2 equals 1 with a remainder of 0.
9      1 divided by 2 equals 0 with a remainder of 1.
10     Now, reading the remainders in reverse order, we get the binary representation: ** 1000 **.
11
12  ### Boundary Values:
13     Test with boundary values where the input number `x` is 0 or 1 and the base is 2 or 3.
14     These scenarios should cover the cases where:
15     - The input number is 0 or 1.
16     - The base is 2 or 3, representing the smallest possible bases.
17
18  | Test Case | Input | Expected Output |
19  |-----|-----|-----|
20  | 1         | `0, 2` | ``'0'``         |
21  | 1         | `0, 3` | ``'0'``         |
22  | 2         | `1, 2` | ``'1'``         |
23  | 2         | `1, 3` | ``'1'``         |
24
25  ### Typical Values:
26     Test with typical inputs where the input number `x` and the base are within reasonable ranges.
27     These could include different combinations of `x` and base values.
28
29  | Test Case | Input | Expected Output |
30  |-----|-----|-----|
31  | 3         | `2, 3` | ``'2'``         |
32  | 4         | `6, 7` | ``'6'``         |
33  | 5         | `8, 3` | ``'22'``        |
34  | 6         | `8, 2` | ``'1000'``      |
35  | 6         | `7, 2` | ``'111'``       |
36  | 6         | `234, 2` | ``'11101010'`` |
37  | 6         | `10, 3` | ``'101'``       |
38
39  ### Special Cases:
40     Consider special cases mentioned or implied in the docstring. For instance:
41     - Test with negative values of `x`.
42     - Test with edge cases where the base is close to the input number.
43
44  | Test Case | Input | Expected Output |
45  |-----|-----|-----|
46  | 7         | `-8, 3` | ``'-22'``       |

```

Figure B.27: Sample 9: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_44.prompt_44 import change_base
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestChangeBase:
13     # Matches with test case 5, 6
14     # The function should correctly convert a given number to a given base.
15     Codiumate: Add more tests
16     def test_convert_number_to_base(self):
17         assert change_base(8, 3) == '22'
18         assert change_base(8, 2) == '1000'
19         assert change_base(7, 2) == '111'
20
21     # Matches with test case 1
22     # The function should return '0' when the input number is 0.
23     Codiumate: Add more tests
24     def test_convert_zero_to_base(self):
25         assert change_base(0, 3) == '0'
26         assert change_base(0, 2) == '0'
```

Figure B.28: Sample 9: Relevant Test Cases

B.10 Sample 10

```

3  def remove_vowels(text):
4      """
5      Assuming that all inputs are valid. Remove_vowels is a function that takes string and returns string without vowels.
6      >>> remove_vowels('')
7      ''
8      >>> remove_vowels("abcdef\nghijklm")
9      'bcdf\nghijklm'
10     >>> remove_vowels('abcdef')
11     'bcdf'
12     >>> remove_vowels('aaaaa')
13     ''
14     >>> remove_vowels('aaBAA')
15     'B'
16     >>> remove_vowels('zbcd')
17     'zbcd'
18     """
19     return "".join([s for s in text if s.lower() not in ["a", "e", "i", "o", "u"]])
20

```

Figure B.29: Sample 10: Function Under Test

```

1  # Function 51 Test Case Criteria {#10}
2
3  ## Test Cases for `remove_vowels` Function
4
5  ### Boundary Values:
6  Test with boundary values where the input text is empty or contains only one character.
7  These scenarios should cover the cases where:
8  - The input text is empty.
9  - The input text contains a single character.
10
11 | Test Case | Input | Expected Output |
12 |-----|-----|-----|
13 | 1 | `''` | `''` |
14 | 2 | ` 'a' ` | ` '' ` |
15 | 3 | ` 'A' ` | ` '' ` |
16
17 ### Typical Values:
18 Test with typical inputs where the input text contains various combinations of vowels and consonants.
19
20 | Test Case | Input | Expected Output |
21 |-----|-----|-----|
22 | 4 | ` 'abcdef' ` | ` 'bcdf' ` |
23 | 4 | ` 'abcdef\nghijklm' ` | ` 'bcdf\nghijklm' ` |
24 | 5 | ` 'zbcd' ` | ` 'zbcd' ` |
25 | 5 | ` 'aaaaa' ` | ` '' ` |
26
27 ### Special Cases:
28 Consider special cases mentioned or implied in the docstring. For instance:
29 - Test with text containing uppercase and lowercase vowels.
30 - Test with text containing uppercase and lowercase consonants.
31 - Test with text containing special characters.
32
33 | Test Case | Input | Expected Output |
34 |-----|-----|-----|
35 | 6 | ` 'aaBAA' ` | ` 'B' ` |
36 | 6 | ` 'aaaaA' ` | ` '' ` |
37 | 6 | ` 'BcDeF' ` | ` 'BcDF' ` |
38 | 7 | ` '123!@#$' ` | ` '123!@#$' ` |
39

```

Figure B.30: Sample 10: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  from experiment_sample_pool.function_51.prompt_51 import remove_vowels
5
6
7  # Generated by CodiumAI
8
9  import pytest
10
11  Codiumate: Add more tests
12  class TestRemoveVowels:
13      # Matches test case 1
14      # Returns empty string when input is empty
15      Codiumate: Add more tests
16      def test_empty_input(self):
17          |   assert remove_vowels('') == ''
18
19      # Matches test case 4
20      # Handles input with only one consonant
21      Codiumate: Add more tests
22      def test_one_consonant(self):
23          |   assert remove_vowels('b') == 'b'
24
25      # Matches test case 6
26      # Handles mixed case input
27      Codiumate: Add more tests
28      def test_handles_mixed_case_input(self):
29          |   assert remove_vowels('aaBAA') == 'B'
30
31      # Matches test case 5
32      # Removes all vowels from input string
33      Codiumate: Add more tests
34      def test_no_vowels_with_special_characters(self):
35          |   assert remove_vowels('zbcd') == 'zbcd'
36
37      # Matches test case 7
38      # Handles input with special characters
39      Codiumate: Add more tests
40      def test_handles_input_with_special_characters(self):
41          |   assert remove_vowels('!@#$$%^&*()') == '!@#$$%^&*()'
42          |   assert remove_vowels('abc!@#$$%^&*()def') == 'bc!@#$$%^&*()df'
43          |   assert remove_vowels('!@#$$%^&*()abc') == '!@#$$%^&*()bc'
44          |   assert remove_vowels('!@#$$%^&*()aeiou') == '!@#$$%^&*()'
45          |   assert remove_vowels('!@#$$%^&*()AEIOU') == '!@#$$%^&*()'
46          |   assert remove_vowels('!@#$$%^&*()zbcd') == '!@#$$%^&*()zbcd'

```

Figure B.31: Sample 10: Relevant Test Cases

B.11 Sample 11

```

Codiumate: Options | Test this function
3  def common(l1: list, l2: list):
4  """Assuming that all inputs are valid. Return sorted unique common elements for two lists.
5  >>> common([1, 4, 3, 34, 653, 2, 5], [5, 7, 1, 5, 9, 653, 121])
6  [1, 5, 653]
7  >>> common([5, 3, 2, 8], [3, 2])
8  [2, 3]
9
10 """
11 ret = set()
12 for e1 in l1:
13     for e2 in l2:
14         if e1 == e2:
15             ret.add(e1)
16 return sorted(list(ret))
17

```

Figure B.32: Sample 11: Function Under Test

```

1  # {#11} Test Case Selection Criteria for Function 58      You, last month • reorganise test case in selecti
2
3  ## Test Cases for `common` Function
4
5  ### Boundary Values:
6  Test with boundary values where one or both lists are empty. These scenarios should cover the cases where:
7  - One or both lists are empty.
8
9  | Test Case | Input                | Expected Output |
10 |-----|-----|-----|
11 | 1          | `[], []`             | `[]`            |
12 | 2          | `[1, 2, 3], []`     | `[]`            |
13 | 3          | `[], [4, 5, 6]`     | `[]`            |
14
15 ### Typical Values:
16 Test with typical inputs where both lists contain various combinations of elements, including duplicates.
17
18 | Test Case | Input                | Expected Output |
19 |-----|-----|-----|
20 | 4          | `[1, 4, 3, 34, 653, 2, 5], [5, 7, 1, 5, 9, 653, 121]` | `[1, 5, 653]`   |
21 | 4          | `[5, 3, 2, 8], [3, 2]` | `[2, 3]`        |
22 | 4          | `[1, 1, 1, 2, 2, 2, 3], [2, 2, 3, 3, 4, 4]` | `[2, 3]`        |
23 | 4          | `[8, 9, 7], [9, 7, 8]` | `[7, 8, 9]`     |
24
25 ### Special Cases:
26 Consider special cases mentioned or implied in the docstring. For instance:
27 - Test with lists containing only one common element.
28 - Test with lists containing all common elements.
29 - Test with lists containing no common elements.
30
31 | Test Case | Input                | Expected Output |
32 |-----|-----|-----|
33 | 5          | `[1, 2, 3], [3]`     | `[3]`           |
34 | 6          | `[1, 2, 3], [4, 5, 6]` | `[]`           |
35 | 7          | `[1, 1, 1], [1, 1, 1]` | `[1]`          |
36

```

Figure B.33: Sample 11: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_58.prompt_58 import common
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestCommon:
13     # Matches test case 4
14     # The function returns a list of sorted unique common elements for two lists.
15     Codiumate: Add more tests
16     def test_sorted_unique_common_elements(self):
17         # Arrange
18         l1 = [1, 4, 3, 34, 653, 2, 5]
19         l2 = [5, 7, 1, 5, 9, 653, 121]
20
21         # Act
22         result = common(l1, l2)
23
24         # Assert
25         assert result == [1, 5, 653]
26
27     # Matches test case 1
28     # The function handles empty lists and returns an empty list.
29     Codiumate: Add more tests
30     def test_empty_lists(self):
31         # Arrange
32         l1 = []
33         l2 = []
34
35         # Act
36         result = common(l1, l2)
37
38         # Assert
39         assert result == []
40
41     # Matches test case 6
42     # The function handles lists with no common elements and returns an empty list.
43     Codiumate: Add more tests
44     def test_handles_lists_with_no_common_elements(self):
45         # Arrange
46         l1 = [1, 2, 3]
47         l2 = [4, 5, 6]
48
49         # Act
50         result = common(l1, l2)
51
52         # Assert
53         assert result == []
```

Figure B.34: Sample 11: Relevant Test Cases

B.12 Sample 12

```
1
2 ✓ FIX = """
3   Add more test cases.
4   """
5
6 Codiumate: Options | Test this function
7 ✓ def vowels_count(s):
8     """Assuming that all inputs are valid. This function does vowels_count which takes a string representing
9     a word as input and returns the number of vowels in the string.
10    Vowels in this case are 'a', 'e', 'i', 'o', 'u'. Here, 'y' is also a
11    vowel, but only when it is at the end of the given word.
12
13    Example:
14    >>> vowels_count("abcde")
15    2
16    >>> vowels_count("ACEDY")
17    3
18    """
19    vowels = "aeiouAEIOU"
20    n_vowels = sum(c in vowels for c in s)
21    if s[-1] == 'y' or s[-1] == 'Y':
22        n_vowels += 1
23    return n_vowels
```

Figure B.35: Sample 12: Function Under Test

```

1  # {#12} Test Case Selection Criteria for Function 64
2
3  ## Test Cases for `vowels_count` Function
4
5  ### Boundary Values:
6  Test with boundary values where the input string is empty or contains only one character.
7  These scenarios should cover the cases where:
8  - The input string is empty.
9  - The input string contains a single character.
10
11 | Test Case | Input | Expected Output |
12 |-----|-----|-----|
13 | 1 | `` | `0` |
14 | 2 | `a` | `1` |
15 | 3 | `Y` | `1` |
16
17 ### Typical Values:
18 Test with typical inputs where the input string contains various combinations of vowels and consonants.
19
20 | Test Case | Input | Expected Output |
21 |-----|-----|-----|
22 | 4 | `abcde` | `2` |
23 | 5 | `ACEDY` | `3` |
24 | 5 | `YAY` | `2` |
25 | 5 | `yay` | `2` |
26 | 6 | `hello` | `2` |
27 | 6 | `banana` | `3` |
28
29 ### Special Cases:
30 Consider special cases mentioned or implied in the docstring. For instance:
31 - Test with input strings containing uppercase and lowercase letters.
32 - Test with input strings containing only vowels.
33 - Test with input strings not containing any vowels.
34
35 | Test Case | Input | Expected Output |
36 |-----|-----|-----|
37 | 7 | `YELLow` | `2` |
38 | 8 | `aeiouy` | `6` |
39 | 9 | `AEIOUY` | `6` |
40 | 9 | `aY` | `2` |
41 | 10 | `xyz` | `0` |
42 | 11 | `BCDXYZ` | `0` |

```

Figure B.36: Sample 12: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  from experiment_sample_pool.function_64.prompt_64 import vowels_count
5
6
7  # Generated by CodiumAI
8
9  import pytest
10
11  Codiumate: Add more tests
12  class TestVowelsCount:
13      # Matches test case 4, 8
14      # Returns the correct number of vowels in a lowercase string
15  def test_lowercase_string(self):
16      assert vowels_count("abcde") == 2
17      assert vowels_count("aeiou") == 5
18
19      # Matches test case 1
20      # Returns 0 for an empty string
21  Codiumate: Add more tests
22  def test_empty_string(self):
23      assert vowels_count("") == 0
24
25      # Duplicate of test case 4, 8
26      # Returns the correct number of vowels in an uppercase string
27  def test_lowercase_string(self):
28      assert vowels_count("abcde") == 2
29      assert vowels_count("aeiou") == 5
30
31      # Matches test case 5
32      # Returns 1 for a string with only one vowel at the end (y or Y)
33  Codiumate: Add more tests
34  def test_string_with_one_vowel_at_end(self):
35      assert vowels_count("abcdey") == 2
36      assert vowels_count("ACEDY") == 3
37
38      # Duplicate of test case 4, 8
39      # Returns the correct number of vowels in a mixed case string
40  Codiumate: Add more tests
41  def test_lowercase_string(self):
42      assert vowels_count("abcde") == 2
43      assert vowels_count("aeiou") == 5
44
45      # Matches test case 10, 11
46      # Returns 0 for a string with no vowels
47  Codiumate: Add more tests
48  def test_no_vowels(self):
49      assert vowels_count("xyz") == 0
50      assert vowels_count("BCDFG") == 0

```

Figure B.37: Sample 12: Relevant Test Cases

B.13 Sample 13

```

Codiumate: Options | Test this function
2 def fruit_distribution(s,n):
3     """
4     Assuming that all inputs are valid. This function will be given a string that represents a number of apples and oranges
5     that are distributed in a basket of fruit this basket contains
6     apples, oranges, and mango fruits. Given the string that represents the total number of
7     the oranges and apples and an integer that represent the total number of the fruits
8     in the basket return the number of the mango fruits in the basket.
9     for example:
10    fruit_distribution("5 apples and 6 oranges", 19) ->19 - 5 - 6 = 8
11    fruit_distribution("0 apples and 1 oranges",3) -> 3 - 0 - 1 = 2
12    fruit_distribution("2 apples and 3 oranges", 100) -> 100 - 2 - 3 = 95
13    fruit_distribution("100 apples and 1 oranges",120) -> 120 - 100 - 1 = 19
14    """
15    lis = list()
16    for i in s.split(' '):
17        if i.isdigit():
18            lis.append(int(i))
19    return n - sum(lis)

```

Figure B.38: Sample 13: Function Under Test

```

1 | # {#13} Test Case Selection Criteria for Function 67
2
3 | ## Test Cases for `fruit_distribution` Function
4
5 | ### Boundary Values:
6 | Test with boundary values where the string `s` represents no fruits or the total number of fruits `n` is minimal.
7
8 | | Test Case | Input | Expected Output |
9 | |-----|-----|-----|
10 | | 1 | | `0 apples and 0 oranges`, 1 | `1` |
11 | | 2 | | `0 apples and 0 oranges`, 0 | `0` |
12
13 | ### Typical Values:
14 | Test with typical inputs where the string `s` represents various combinations of apples and oranges, and `n` is typical.
15
16 | | Test Case | Input | Expected Output |
17 | |-----|-----|-----|
18 | | 3 | | `5 apples and 6 oranges`, 19 | `8` |
19 | | 3 | | `2 apples and 3 oranges`, 100 | `95` |
20
21 | ### Special Cases:
22 | Consider special cases mentioned or implied in the docstring. For instance:
23 | - Test with no apples or no oranges in the input string.
24 | - Test with a large number of fruits in the basket.
25
26 | | Test Case | Input | Expected Output |
27 | |-----|-----|-----|
28 | | 4 | | `0 apples and 10 oranges`, 15 | `5` |
29 | | 5 | | `20 apples and 0 oranges`, 100 | `80` |
30

```

Figure B.39: Sample 13: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_67.prompt_67 import fruit_distribution
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestFruitDistribution:
13     # Matches test case 2
14     # The function returns 0 when given a string that represents 0 apples and 0 oranges
15     # and an integer that represents the total number of fruits in the basket.
16     Codiumate: Add more tests
17     def test_returns_zero_when_given_zero_fruits(self):
18         result = fruit_distribution("0 apples and 0 oranges", 0)
19         assert result == 0
20
21     # Matches test case 3, 4
22     # The function returns the correct number of mango fruits in the basket when given
23     # a string that represents the total number of oranges and apples and an integer that
24     # represents the total number of fruits in the basket,
25     # even if the string contains additional words or characters.
26     Codiumate: Add more tests
27     def test_returns_correct_number_of_mango_fruits(self):
28         result = fruit_distribution("5 apples and 6 oranges", 19)
29         assert result == 8
30
31         result = fruit_distribution("0 apples and 1 oranges", 3)
32         assert result == 2
33
34         result = fruit_distribution("2 apples and 3 oranges", 100)
35         assert result == 95
36
37         result = fruit_distribution("100 apples and 1 oranges", 120)
38         assert result == 19
39
```

Figure B.40: Sample 13: Relevant Test Cases

B.14 Sample 14

```
Codiumate: Options | Test this function
2  def is_simple_power(x, n):
3      """Assuming that all inputs are valid.
4      This function returns true if a number x is a simple
5      power of n and false in other cases.
6      x is a simple power of n if n**int=x
7      For example:
8      is_simple_power(1, 4) => true
9      is_simple_power(2, 2) => true
10     is_simple_power(8, 2) => true
11     is_simple_power(3, 2) => false
12     is_simple_power(3, 1) => false
13     is_simple_power(5, 3) => false
14     """
15     if (n == 1):
16         return (x == 1)
17     power = 1
18     while (power < x):
19         power = power * n
20     return (power == x)
```

Figure B.41: Sample 14: Function Under Test

```

1  # {#14} Test Case Selection Criteria for Function 76
2
3  ## Test Cases for `is_simple_power` Function
4
5  ### Boundary Values:
6  Test with boundary values where `x` and `n` are minimal or maximal.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1          | `1, 1` | `True`          |
11 | 1          | `2, 2` | `True`          |
12 | 2          | `1, 2` | `True`          |
13 | 3          | `2, 1` | `False`         |
14 | 4          | `2, 3` | `False`         |
15
16 ### Typical Values:
17 Test with typical inputs where `x` is raised to the power of `n` and vice versa.
18
19 | Test Case | Input | Expected Output |
20 |-----|-----|-----|
21 | 5          | `8, 2` | `True`          |
22 | 5          | `9, 3` | `True`          |
23 | 5          | `16, 2` | `True`         |
24 | 6          | `3, 2` | `False`         |
25 | 6          | `5, 3` | `False`         |
26
27
28 ### Special Cases:
29 Consider special cases mentioned or implied in the docstring. For instance:
30 - Test with `x` and `n` both being 0.
31
32 | Test Case | Input | Expected Output |
33 |-----|-----|-----|
34 | 7          | `0, 0` | `True`          |
35

```

Figure B.42: Sample 14: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_76.prompt_76 import is_simple_power
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestIsSimplePower:
13     # Matches test case 1, 2, 5
14     # Returns True for simple powers of n
15     Codiumate: Add more tests
16     def test_simple_power(self):
17         assert is_simple_power(1, 4) == True
18         assert is_simple_power(2, 2) == True
19         assert is_simple_power(8, 2) == True
20
21     # Matches test case 6, 3, 6
22     # Returns False for non-simple powers of n
23     Codiumate: Add more tests
24     def test_non_simple_powers(self):
25         assert is_simple_power(3, 2) == False
26         assert is_simple_power(3, 1) == False
27         assert is_simple_power(5, 3) == False
```

Figure B.43: Sample 14: Relevant Test Cases

B.15 Sample 15

```
Codiumate: Options | Test this function
3  ✓ def iscube(a):
4  ✓     '''
5     Assuming that all inputs are valid.
6     This function takes an integer a and returns True
7     if this integer is a cube of some integer number.
8     Note: you may assume the input is always valid.
9     Examples:
10    iscube(1) ==> True
11    iscube(2) ==> False
12    iscube(-1) ==> True
13    iscube(64) ==> True
14    iscube(0) ==> True
15    iscube(180) ==> False
16    '''
17    a = abs(a)
18    return int(round(a ** (1. / 3))) ** 3 == a
19
```

Figure B.44: Sample 15: Function Under Test

```

1  # {#15} Test Case Selection Criteria for Function 77
2
3  ## Test Cases for `iscube` Function
4
5  ### Boundary Values: *To double check if it should be other type of test case.
6  Test with boundary values where `a` is minimal or maximal.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1         | `1`  | `True`          |
11 | 2         | `0`  | `True`          |
12 | 3         | `2`  | `False`         |
13
14 ### Typical Values:
15 Test with typical inputs where `a` is or is not a cube of some integer number.
16
17 | Test Case | Input | Expected Output |
18 |-----|-----|-----|
19 | 4         | `100`| `False`         |
20 | 4         | `180`| `False`         |
21 | 5         | `8`  | `True`          |
22 | 5         | `125`| `True`          |
23 | 5         | `64` | `True`          |
24 | 5         | `0`  | `True`          |
25 | 5         | `1000`| `True`         |
26 | 6         | `8888`| `False`        |
27
28 ### Special Cases:
29 Consider special cases mentioned or implied in the docstring. For instance:
30 - Test with a negative input.
31
32 | Test Case | Input | Expected Output |
33 |-----|-----|-----|
34 | 7         | `-1` | `True`          |

```

Figure B.45: Sample 15: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 from experiment_sample_pool.function_77.prompt_77 import iscube
5
6
7 # Generated by CodiumAI
8
9 import pytest
10
11 Codiumate: Add more tests
12 class TestIscube:
13     # Matches test case 1
14     # Returns True for the smallest possible perfect cube
15     Codiumate: Add more tests
16     def test_smallest_perfect_cube(self):
17         | assert iscube(1) == True
18
19     # Matches test case 5
20     # Returns True for the largest possible perfect cube
21     Codiumate: Add more tests
22     def test_returns_true_for_largest_possible_perfect_cube(self):
23         | assert iscube(2147483647) == True
24
25     # Matches test case 3
26     # Returns False for the smallest possible non-cube number
27     Codiumate: Add more tests
28     def test_returns_false_for_smallest_non_cube_number(self):
29         | assert iscube(2) == False
30
31     # Matches test case 6
32     # Returns False for the largest possible non-cube number
33     Codiumate: Add more tests
34     def test_returns_false_for_largest_non_cube_number(self):
35         | assert iscube(999999999) == False
36
37     # Matches test case 7
38     # Returns False for a negative non-cube number
39     Codiumate: Add more tests
40     def test_negative_non_cube_number(self):
41         | assert iscube(-2) == False
42
43     # Matches test case 1, 7, 5, 2
44     # Returns True for a perfect cube
45     Codiumate: Add more tests
46     def test_perfect_cube(self):
47         | assert iscube(1) == True
48         | assert iscube(-1) == True
49         | assert iscube(64) == True
50         | assert iscube(0) == True
51
52     # Duplicate test case 3
53     # Returns False for a non-cube number
54     Codiumate: Add more tests
55     def test_returns_false_for_non_cube_number(self):
56         | assert iscube(2) == False

```

Figure B.46: Sample 15: Relevant Test Cases

B.16 Sample 16

```

Codumate: Options | Test this function
2 def hex_key(num):
3     """Assuming that all inputs are valid. This function receives a hexadecimal number as a string and counts the number of hexadecimal
4     digits that are primes (prime number, or a prime, is a natural number
5     greater than 1 that is not a product of two smaller natural numbers).
6     Hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
7     Prime numbers are 2, 3, 5, 7, 11, 13, 17,...
8     So you have to determine a number of the following digits: 2, 3, 5, 7,
9     B (=decimal 11), D (=decimal 13).
10    Note: you may assume the input is always correct or empty string,
11    and symbols A,B,C,D,E,F are always uppercase.
12    Examples:
13    For num = "AB" the output should be 1.
14    For num = "1077E" the output should be 2.
15    For num = "ABED1A33" the output should be 4.
16    For num = "123456789ABCDEF0" the output should be 6.
17    For num = "2020" the output should be 2.
18    """
19    primes = ('2', '3', '5', '7', 'B', 'D')
20    total = 0
21    for i in range(0, len(num)):
22        if num[i] in primes:
23            total += 1
24    return total

```

Figure B.47: Sample 16: Function Under Test

```

1 # {#16} Test Case Selection Criteria for Function 78
2
3 ## Test Cases for `hex_key` Function
4
5 ### Boundary Values:
6 Test with boundary values where `num` is minimal or maximal.
7
8 | Test Case | Input | Expected Output |
9 |-----|-----|-----|
10 | 1 | `` | `0`
11 | 2 | `0` | `0`
12 | 3 | `F` | `0`
13 | 4 | `2F` | `1`
14
15 ### Typical Values:
16 Test with typical inputs where `num` contains various hexadecimal digits.
17
18 | Test Case | Input | Expected Output |
19 |-----|-----|-----|
20 | 4 | `AB` | `1`
21 | 5 | `1077E` | `2`
22 | 5 | `ABED1A33` | `4`
23 | 5 | `2020` | `2`
24 | 6 | `123456789ABCDEF0` | `6`
25
26
27 ### Special Cases:
28 Consider special cases mentioned or implied in the docstring. For instance:
29 - Test with an input containing only one prime hexadecimal digit.
30
31 | Test Case | Input | Expected Output |
32 |-----|-----|-----|
33 | 7 | `5` | `1`

```

Figure B.48: Sample 16: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  # Generated by CodiumAI
5  from experiment_sample_pool.function_78.prompt_78 import hex_key
6
7
8  import pytest
9
10  Codiumate: Add more tests
11  class TestHexKey:
12      # Matches test case 1
13      # Should return 0 for empty string input
14      Codiumate: Add more tests
15      def test_empty_string_input(self):
16          |   assert hex_key("") == 0
17
18      # Matches test case 2
19      # Should return 0 for input "0"
20      Codiumate: Add more tests
21      def test_input_zero(self):
22          |   assert hex_key("0") == 0
23
24      # Matches test case 4
25      # Should return 1 for input "AB"
26      Codiumate: Add more tests
27      def test_return_1_for_input_AB(self):
28          |   assert hex_key("AB") == 1
29
30      # Matches test case 5
31      # Should return 2 for input "1077E"
32      Codiumate: Add more tests
33      def test_return_2_for_input_1077E(self):
34          |   assert hex_key("1077E") == 2
35
36      # Matches test case 5
37      # Should return 4 for input "ABED1A33"
38      Codiumate: Add more tests
39      def test_hex_key_input_ABED1A33(self):
40          |   assert hex_key("ABED1A33") == 4
41
42      # Matches test case 6
43      # Should return 6 for input "123456789ABCDEF0"
44      Codiumate: Add more tests
45      def test_hex_key_returns_six_for_input_123456789ABCDEF0(self):
46          |   assert hex_key("123456789ABCDEF0") == 6
47
48      # Matches test case 5
49      # Should return 2 for input "2020"
50      Codiumate: Add more tests
51      def test_return_2_for_input_2020(self):
52          |   assert hex_key("2020") == 2
53
54

```

Figure B.49: Sample 16: Relevant Test Cases, part 1/3

```
46
47     # Matches test case 7
48     # Should return 1 for input "2"
49     Codiumate: Add more tests
50     def test_return_1_for_input_2(self):
51         |     assert hex_key("2") == 1
52
53     # Matches test case 7
54     # Should return 1 for input "3"
55     Codiumate: Add more tests
56     def test_return_1_for_input_3(self):
57         |     assert hex_key("3") == 1
58
59     # Matches test case 7
60     # Should return 1 for input "5"
61     Codiumate: Add more tests
62     def test_return_1_for_input_5(self):
63         |     assert hex_key("5") == 1
64
65     # Matches test case 7
66     # Should return 1 for input "7"
67     Codiumate: Add more tests
68     def test_return_1_for_input_7(self):
69         |     assert hex_key("7") == 1
70
71     # Matches test case 7
72     # Should return 1 for input "8"
73     Codiumate: Add more tests
74     def test_return_1_for_input_B(self):
75         |     assert hex_key("B") == 1
76
77     # Matches test case 7
78     # Should return 1 for input "D"
79     Codiumate: Add more tests
80     def test_return_1_for_input_D(self):
81         |     assert hex_key("D") == 1
82
83     # Matches test case 3
84     # Should return 0 for input "A"
85     Codiumate: Add more tests
86     def test_return_0_for_input_A(self):
87         |     assert hex_key("A") == 0
88
89     # Matches test case 3
90     # Should return 0 for input "F"
91     Codiumate: Add more tests
92     def test_return_0_for_input_F(self):
93         |     assert hex_key("F") == 0
94
95     # Matches test case 3
96     # Should return 0 for input "E"
97     Codiumate: Add more tests
98     def test_return_zero_for_input_E(self):
99         |     assert hex_key("E") == 0
```

Figure B.50: Sample 16: Relevant Test Cases, part 2/3

```
92     # Matches test case 3
93     # Should return 0 for input "C"
    Codiumate: Add more tests
94     def test_return_0_for_input_C(self):
95         |     assert hex_key("C") == 0
96
97     # Matches test case 3
98     # Should return 0 for input "1"
    Codiumate: Add more tests
99     def test_return_0_for_input_1(self):
100        |     assert hex_key("1") == 0
101
102    # Matches test case 3
103    # Should return 0 for input "4"
    Codiumate: Add more tests
104    def test_return_0_for_input_4(self):
105        |     assert hex_key("4") == 0
106
107    # Matches test case 3
108    # Should return 0 for input "6"
    Codiumate: Add more tests
109    def test_return_zero_for_input_six(self):
110        |     assert hex_key("6") == 0
```

Figure B.51: Sample 16: Relevant Test Cases, part 3/3

B.17 Sample 17

```

Codiumate: Options | Test this function
2  def decimal_to_binary(decimal):
3      """Assuming that all inputs are valid. You will be given a number in decimal form and your task is to convert it to
4      binary format. The function should return a string, with each character representing a binary
5      number. Each character in the string will be '0' or '1'.
6
7      There will be an extra couple of characters 'db' at the beginning and at the end of the string.
8      The extra characters are there to help with the format.
9
10     Examples:
11     decimal_to_binary(15) # returns "db1111db"
12     decimal_to_binary(32) # returns "db10000db"
13     """
14     return "db" + bin(decimal)[2:] + "db"
15

```

Figure B.52: Sample 17: Function Under Test

```

1  # {#17} Test Case Selection Criteria for decimal_to_binary Function 79    You, 3 seconds ago
2
3  ## Test Cases for `decimal_to_binary` Function
4
5  ### Boundary Values:
6  Test with boundary values where `decimal` is minimal or maximal.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1         | `0`   | `db0db`         |
11 | 2         | `1`   | `db1db`         |
12 | 3         | `255` | `db11111111db` |
13 | 3         | `256` | `db10000000db` |
14
15 ### Typical Values:
16 Test with typical inputs where `decimal` is a number between the minimal and maximal range.
17
18 | Test Case | Input | Expected Output |
19 |-----|-----|-----|
20 | 4         | `15`  | `db1111db`      |
21 | 4         | `32`  | `db10000db`     |
22 | 4         | `100` | `db1100100db`   |
23
24
25 ### Special Cases:
26 Consider special cases mentioned or implied in the docstring. For instance:
27 - Test with a large number.
28 - Test with cases where `decimal` is negative.
29
30 | Test Case | Input | Expected Output |
31 |-----|-----|-----|
32 | 5         | `-15` | `dbb1111db`     |
33 | 5         | `-32` | `dbb10000db`    |
34 | 6         | `123456` | `db1111000100100000db` |

```

Figure B.53: Sample 17: Selection Criteria


```
58     # Matches test case 5
59     # Should handle input values close to the minimum integer value (e.g. -2147483648)
    Codiumate: Add more tests
60     def test_decimal_to_binary_with_minimum_integer_value(self):
61         assert decimal_to_binary(-2147483648) == "db100000000000000000000000000000db"
62
63     # Matches test case 6
64     # Should handle input values close to the maximum integer value (e.g. 2147483647)
    Codiumate: Add more tests
65     def test_decimal_to_binary_with_large_input(self):
66         assert decimal_to_binary(2147483647) == "db11111111111111111111111111111111db"
67         assert decimal_to_binary(2147483646) == "db11111111111111111111111111111110db"
68         assert decimal_to_binary(2147483645) == "db11111111111111111111111111111101db"
69         assert decimal_to_binary(2147483644) == "db11111111111111111111111111111100db"
70         assert decimal_to_binary(2147483643) == "db11111111111111111111111111111101db"
71         assert decimal_to_binary(2147483642) == "db11111111111111111111111111111101db"
72         assert decimal_to_binary(2147483641) == "db11111111111111111111111111111001db"
73         assert decimal_to_binary(2147483640) == "db11111111111111111111111111111000db"
74         assert decimal_to_binary(2147483639) == "db11111111111111111111111111110111db"
75         assert decimal_to_binary(2147483638) == "db11111111111111111111111111110110db"
```

Figure B.55: Sample 17: Relevant Test Cases, part 2/2

B.18 Sample 18

```

1
2 Codiumate: Options | Test this function
3 def numerical_letter_grade(grades):
4     """Assuming that all inputs are valid. It is the last week of the semester and the teacher has to give the grades
5     to students. The teacher has been making her own algorithm for grading.
6     The only problem is, she has lost the code she used for grading.
7     She has given you a list of GPAs for some students and this function can output a list of letter grades using the following table:
8
9     GPA | Letter grade
10    > 3.7 | A+
11    > 3.3 | A-
12    > 3.0 | B+
13    > 2.7 | B
14    > 2.3 | B-
15    > 2.0 | C+
16    > 1.7 | C
17    > 1.3 | C-
18    > 1.0 | D+
19    > 0.7 | D
20    > 0.0 | D-
21    > 0.0 | E
22
23 Example:
24 grade_equation([4.0, 3, 1.7, 2, 3.5]) ==> ['A+', 'B', 'C-', 'C', 'A-']
25 """
26
27 letter_grade = []
28 for gpa in grades:
29     if gpa == 4.0:
30         letter_grade.append("A+")
31     elif gpa > 3.7:
32         letter_grade.append("A")
33     elif gpa > 3.3:
34         letter_grade.append("A-")
35     elif gpa > 3.0:
36         letter_grade.append("B+")
37     elif gpa > 2.7:
38         letter_grade.append("B")
39     elif gpa > 2.3:
40         letter_grade.append("B-")
41     elif gpa > 2.0:
42         letter_grade.append("C+")
43     elif gpa > 1.7:
44         letter_grade.append("C")
45     elif gpa > 1.3:
46         letter_grade.append("C-")
47     elif gpa > 1.0:
48         letter_grade.append("D+")
49     elif gpa > 0.7:
50         letter_grade.append("D")
51     elif gpa > 0.0:
52         letter_grade.append("D-")
53     else:
54         letter_grade.append("E")
55 return letter_grade

```

Figure B.56: Sample 18: Function Under Test

```

1  # {#18} Test Case Selection Criteria for numerical_letter_grade Function 81
2
3  ## Test Cases for `numerical_letter_grade` Function
4
5  ### Boundary Values:
6  Test with boundary values where `grades` contains the minimum and maximum GPAs.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1 | `[0.0]` | `['E']` |
11 | 2 | `[0.7]` | `['D-']` |
12 | 2 | `[1.0]` | `['D']` |
13 | 2 | `[1.3]` | `['D+']` |
14 | 2 | `[1.7]` | `['C-']` |
15 | 2 | `[2.7]` | `['B-']` |
16 | 2 | `[3.0]` | `['B']` |
17 | 2 | `[3.7]` | `['A-']` |
18 | 3 | `[4.0]` | `['A+']` |
19
20 ### Typical Values:
21 Test with typical inputs where `grades` contains various GPAs.
22
23 | Test Case | Input | Expected Output |
24 |-----|-----|-----|
25 | 4 | `[0.6]` | `['D-']` |
26 | 4 | `[0.8]` | `['D']` |
27 | 5 | `[4.0, 3, 1.7, 2, 3.5]` | `['A+', 'B', 'C-', 'C+', 'A-']` |
28 | 5 | `[2.3, 3.7, 1.0, 2.7]` | `['B-', 'A', 'D+', 'B']` |
29 | 5 | `[1.3, 0.7, 3.3, 1.7]` | `['C-', 'D', 'A-', 'C']` |
30 | 6 | `[0.1, 0.3, 0.7]` | `['D-', 'D-', 'D-']` |
31 | 7 | `[0.8, 0.9, 1.0]` | `['D', 'D', 'D']` |
32 | 8 | `[1.1, 1.2, 1.3]` | `['D+', 'D+', 'D+']` |
33 | 9 | `[1.4, 1.5, 1.6]` | `['C-', 'C-', 'C-']` |
34 | 10 | `[1.8, 1.9, 2.0]` | `['C', 'C', 'C']` |
35 | 11 | `[2.1, 2.2, 2.3]` | `['C+', 'C+', 'C+']` |
36 | 12 | `[2.4, 2.5, 2.6]` | `['B-', 'B-', 'B-']` |
37 | 13 | `[2.8, 2.9, 3.0]` | `['B', 'B', 'B']` |
38 | 14 | `[3.1, 3.2, 3.3]` | `['B+', 'B+', 'B+']` |
39 | 15 | `[3.4, 3.5, 3.6]` | `['A-', 'A-', 'A-']` |
40 | 16 | `[3.7, 3.8, 3.9]` | `['A', 'A', 'A']` |
41
42
43 ### Special Cases:
44 Consider special cases mentioned or implied in the docstring. For instance:
45 - Test with an empty list.
46
47 | Test Case | Input | Expected Output |
48 |-----|-----|-----|
49 | 17 | `[]` | `[]` |
50

```

Figure B.57: Sample 18: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  # Generated by CodiumAI
5  from experiment_sample_pool.function_81.prompt_81 import numerical_letter_grade
6
7
8  import pytest
9
10  Codiumate: Add more tests
11  class TestNumericalLetterGrade:
12      # Matches test case 5
13      # The function correctly converts a list of GPAs to a list of letter grades.
14      Codiumate: Add more tests
15      def test_convert_gpas_to_letter_grades(self):
16          # Arrange
17          gpas = [4.0, 3, 1.7, 2, 3.5]
18          expected_result = ['A+', 'B', 'C-', 'C', 'A-']
19
20          # Act
21          result = numerical_letter_grade(gpas)
22
23          # Assert
24          assert result == expected_result
25
26      # Matches test case 6
27      # The function correctly handles an empty list of GPAs.
28      Codiumate: Add more tests
29      def test_handle_empty_gpa_list(self):
30          # Arrange
31          gpas = []
32          expected_result = []
33
34          # Act
35          result = numerical_letter_grade(gpas)
36
37          # Assert
38          assert result == expected_result
39
40      # Matches test case 3
41      # The function returns the correct letter grade for a GPA of 4.0.
42      Codiumate: Add more tests
43      def test_correct_letter_grade_for_4_0_gpa(self):
44          # Arrange
45          gpas = [4.0]
46          expected_result = ['A+']
47
48          # Act
49          result = numerical_letter_grade(gpas)
50
51          # Assert
52          assert result == expected_result

```

Figure B.58: Sample 18: Relevant Test Cases, part 1/5

```
50
51 # Matches test case 14
52 # The function returns the correct letter grade for a GPA between 3.7 and 4.0.
  Codiumate: Add more tests
53 def test_correct_letter_grade_for_gpa_between_3_7_and_4_0(self):
54     # Arrange
55     gpas = [4.0, 3.8, 3.9, 3.7]
56     expected_result = ['A+', 'A', 'A', 'A']
57
58     # Act
59     result = numerical_letter_grade(gpas)
60
61     # Assert
62     assert result == expected_result
63
64 # Matches test case 15
65 # The function returns the correct letter grade for a GPA between 3.3 and 3.7.
  Codiumate: Add more tests
66 def test_correct_letter_grade_for_gpa_between_3_3_and_3_7(self):
67     # Arrange
68     gpas = [3.4, 3.5, 3.6]
69     expected_result = ['A-', 'A-', 'A-']
70
71     # Act
72     result = numerical_letter_grade(gpas)
73
74     # Assert
75     assert result == expected_result
76
77 # Matches test case 14
78 # The function returns the correct letter grade for a GPA between 3.0 and 3.3.
  Codiumate: Add more tests
79 def test_correct_letter_grade_for_gpa_between_3_and_3_3(self):
80     # Arrange
81     gpas = [3.1, 3.2, 3.3]
82     expected_result = ['B', 'B', 'B-']
83
84     # Act
85     result = numerical_letter_grade(gpas)
86
87     # Assert
88     assert result == expected_result
89
90 # Matches test case 12
91 # The function returns the correct letter grade for a GPA between 2.3 and 2.7.
  Codiumate: Add more tests
92 def test_correct_letter_grade_for_gpa_between_2_3_and_2_7(self):
93     # Arrange
94     gpas = [2.4, 2.5, 2.6]
95     expected_result = ['B-', 'B-', 'B-']
96
97     # Act
98     result = numerical_letter_grade(gpas)
99
100     # Assert
101     assert result == expected_result
102
```

Figure B.59: Sample 18: Relevant Test Cases, part 2/5

```
103 # Matches test case 13
104 # The function returns the correct letter grade for a GPA between 2.7 and 3.0.
    Codiumate: Add more tests
105 def test_correct_letter_grade_for_gpa_between_2_7_and_3_0(self):
106     # Arrange
107     gpas = [2.8, 2.9, 3.0]
108     expected_result = ['B', 'B', 'B+']
109
110     # Act
111     result = numerical_letter_grade(gpas)
112
113     # Assert
114     assert result == expected_result
115
116 # Matches test case 1
117 # The function returns the correct letter grade for a GPA of 0.0.
    Codiumate: Add more tests
118 def test_correct_letter_grade_for_gpa_0(self):
119     # Arrange
120     gpas = [0.0]
121     expected_result = ['E']
122
123     # Act
124     result = numerical_letter_grade(gpas)
125
126     # Assert
127     assert result == expected_result
128
129 # Matches test case 6
130 # The function returns the correct letter grade for a GPA between 0.7 and 1.0.
    Codiumate: Add more tests
131 def test_correct_letter_grade_for_gpa_between_0_7_and_1_0(self):
132     # Arrange
133     gpas = [0.8, 0.9, 1.0]
134     expected_result = ['D-', 'D-', 'D-']
135
136     # Act
137     result = numerical_letter_grade(gpas)
138
139     # Assert
140     assert result == expected_result
141
142 # Matches test case 6
143 # The function returns the correct letter grade for a GPA between 0.0 and 0.7.
    Codiumate: Add more tests
144 def test_correct_letter_grade_for_gpa_between_0_and_0_7(self):
145     # Arrange
146     gpas = [0.0, 0.5, 0.7]
147     expected_result = ['E', 'D-', 'D-']
148
149     # Act
150     result = numerical_letter_grade(gpas)
151
152     # Assert
153     assert result == expected_result
```

Figure B.60: Sample 18: Relevant Test Cases, part 3/5

```
155 # Matches test case 11
156 # The function returns the correct letter grade for a GPA between 2.0 and 2.3.
    Codiumate: Add more tests
157 def test_correct_letter_grade_for_gpa_between_2_and_2_3(self):
158     # Arrange
159     gpas = [2.1, 2.2, 2.3]
160     expected_result = ['C+', 'C+', 'C-']
161
162     # Act
163     result = numerical_letter_grade(gpas)
164
165     # Assert
166     assert result == expected_result
167
168 # Matches test case 9
169 # The function returns the correct letter grade for a GPA between 1.3 and 1.7.
    Codiumate: Add more tests
170 def test_correct_letter_grade_for_gpa_between_1_3_and_1_7(self):
171     # Arrange
172     gpas = [1.4, 1.5, 1.6]
173     expected_result = ['C-', 'C-', 'C-']
174
175     # Act
176     result = numerical_letter_grade(gpas)
177
178     # Assert
179     assert result == expected_result
180
181 # Matches test case 10
182 # The function returns the correct letter grade for a GPA between 1.7 and 2.0.
    Codiumate: Add more tests
183 def test_correct_letter_grade_for_gpa_between_1_7_and_2_0(self):
184     # Arrange
185     gpas = [1.8, 1.9, 2.0]
186     expected_result = ['C-', 'C-', 'C+']
187
188     # Act
189     result = numerical_letter_grade(gpas)
190
191     # Assert
192     assert result == expected_result
193
194 # Matches test case 8
195 # The function returns the correct letter grade for a GPA between 1.0 and 1.3.
    Codiumate: Add more tests
196 def test_correct_letter_grade_for_gpa_between_1_and_1_3(self):
197     # Arrange
198     gpas = [1.0, 1.1, 1.2, 1.3]
199     expected_result = ['D+', 'D+', 'D+', 'C-']
200
201     # Act
202     result = numerical_letter_grade(gpas)
203
204     # Assert
205     assert result == expected_result
```

Figure B.61: Sample 18: Relevant Test Cases, part 4/5

```
206
207     # Matches test case 4
208     # The function correctly handles a list of GPAs with only one element.
    Codiumate: Add more tests
209     def test_single_gpa(self):
210         # Arrange
211         gpas = [3.5]
212         expected_result = ['A-']
213
214         # Act
215         result = numerical_letter_grade(gpas)
216
217         # Assert
218         assert result == expected_result
219
220     # The function correctly handles a list of GPAs with all the same value.
    Codiumate: Add more tests
221     def test_handles_list_of_same_GPAs(self):
222         # Arrange
223         gpas = [3.5, 3.5, 3.5, 3.5]
224         expected_result = ['A-', 'A-', 'A-', 'A-']
225
226         # Act
227         result = numerical_letter_grade(gpas)
228
229         # Assert
230         assert result == expected_result
231
232     # Matches test case 6
233     # The function correctly handles a list of GPAs with all values below 1.0.
    Codiumate: Add more tests
234     def test_handles_list_of_GPAs_below_1(self):
235         # Arrange
236         gpas = [0.5, 0.2, 0.8, 0.1]
237         expected_result = ['D-', 'E', 'D-', 'E']
238
239         # Act
240         result = numerical_letter_grade(gpas)
241
242         # Assert
243         assert result == expected_result
```

Figure B.62: Sample 18: Relevant Test Cases, part 5/5

B.19 Sample 19

```
Codiumate: Options | Test this function
2  def prime_length(string):
3      """Assuming that all inputs are valid. This function takes a string and returns True if the string
4         length is a prime number or False otherwise
5         Examples
6         prime_length('Hello') == True
7         prime_length('abcdcba') == True
8         prime_length('kittens') == True
9         prime_length('orange') == False
10        """
11        l = len(string)
12        if l == 0 or l == 1:
13            return False
14        for i in range(2, l):
15            if l % i == 0:
16                return False
17        return True
18
```

Figure B.63: Sample 19: Function Under Test

```

1  # {#19} Test Case Selection Criteria for prime_length Function 82
2
3  ## Test Cases for `prime_length` Function
4
5  ### Boundary Values:
6  Test with boundary values where the length of `string` is minimal or maximal.
7
8  | Test Case | Input      | Expected Output |
9  |-----|-----|-----|
10 | 1         | ``        | `False`         |
11 | 2         | `a`       | `False`         |
12 | 3         | `ab`      | `True`          |
13 | 3         | `abc`     | `True`          |
14 | 3         | `12345`   | `True`          |
15 | 3         | `1234567` | `True`          |
16 | 4         | `abcdefgh`| `False`         |
17 | 4         | `abcdefghi`| `False`         |
18
19 ### Typical Values:
20 Test with typical inputs where the length of `string` is a prime number or not.
21
22 | Test Case | Input      | Expected Output |
23 |-----|-----|-----|
24 | 3         | `Hello`   | `True`          |
25 | 3         | `abcdcba` | `True`          |
26 | 3         | `kittens` | `True`          |
27 | 4         | `orange`  | `False`         |
28 | 3         | `123456789`| `False`         |
29
30 ### Special Cases:
31 Consider special cases mentioned or implied in the docstring. For instance:
32 - Test with an empty string.
33 - Test with a large string size.
34
35 | Test Case | Input      | Expected Output |
36 |-----|-----|-----|
37 | 1         | ``        | `False`         |
38 | 5         | `0123456789012345678901234567890` | `True`          |

```

Figure B.64: Sample 19: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  # Generated by CodiumAI
5  from experiment_sample_pool.function_82.prompt_82 import prime_length
6
7
8  import pytest
9
10  Codiumate: Add more tests
11  class TestPrimeLength:
12      # Matches test case 3
13      # Returns True for a string with a prime length
14      Codiumate: Add more tests
15      def test_prime_length_prime_length(self):
16          assert prime_length('Hello') == True
17          assert prime_length('abcdcba') == True
18          assert prime_length('kittens') == True
19          assert prime_length('orange') == False
20
21      # Matches test case 3, 4
22      # Returns True for a string with length 3
23      Codiumate: Add more tests
24      def test_prime_length_length_3(self):
25          assert prime_length('abc') == True
26          assert prime_length('xyz') == True
27          assert prime_length('123') == True
28          assert prime_length('apple') == False
29
30      # Matches test case 4
31      # Returns False for a string with a non-prime length
32      Codiumate: Add more tests
33      def test_returns_false_for_non_prime_length(self):
34          assert prime_length('Hello') == False
35          assert prime_length('abcdcba') == False
36          assert prime_length('kittens') == False
37          assert prime_length('orange') == False
38
39      # Matches test case 1
40      # Returns False for an empty string
41      Codiumate: Add more tests
42      def test_empty_string(self):
43          assert prime_length('') == False
44
45      # Matches test case 2
46      # Returns False for a string with length 1
47      Codiumate: Add more tests
48      def test_returns_false_for_string_with_length_1(self):
49          assert prime_length('a') == False
50
51      # Matches test case 3
52      # Returns True for a string with length 2
53      Codiumate: Add more tests
54      def test_returns_true_for_string_with_length_2(self):
55          assert prime_length('ab') == True
56
57

```

Figure B.65: Sample 19: Relevant Test Cases, part 1/2

```

51     # Matches test case 4
52     # Returns False for a string with length 4
    Codiumate: Add more tests
53     def test_returns_false_for_string_with_length_4(self):
54         |     assert prime_length('abcd') == False
55
56     # Matches test case 3
57     # Returns True for a string with length 5
    Codiumate: Add more tests
58     def test_returns_true_for_string_with_length_5(self):
59         |     assert prime_length('aaaaa') == True
60
61     # Matches test case 4
62     # Returns False for a string with length 6
    Codiumate: Add more tests
63     def test_returns_false_for_string_with_length_6(self):
64         |     assert prime_length('abcdef') == False
65
66     # Matches test case 3
67     # Returns True for a string with length 7
    Codiumate: Add more tests
68     def test_returns_true_for_string_with_length_7(self):
69         |     assert prime_length('abcdefg') == True
70
71     # Matches test case 4
72     # Returns False for a string with length 8
    Codiumate: Add more tests
73     def test_returns_false_for_string_with_length_8(self):
74         |     assert prime_length('abcdefgh') == False
75
76     # Matches test case 3
77     # Returns True for a string with length 11
    Codiumate: Add more tests
78     def test_returns_true_for_string_with_length_11(self):
79         |     assert prime_length('aaaaaaaaaaa') == True
80
81     # Matches test case 4
82     # Returns False for a string with length 12
    Codiumate: Add more tests
83     def test_returns_false_for_string_with_length_12(self):
84         |     assert prime_length('Hello') == True
85         |     assert prime_length('abcdcba') == True
86         |     assert prime_length('kittens') == True
87         |     assert prime_length('orange') == False
88         |     assert prime_length('abcdefghijkl') == False
89
90     # Matches test case 3
91     # Returns True for a string with length 13
    Codiumate: Add more tests
92     def test_returns_true_for_string_with_length_13(self):
93         |     assert prime_length('Hello') == True
94         |     assert prime_length('abcdcba') == True
95         |     assert prime_length('kittens') == True
96         |     assert prime_length('orange') == False

```

Figure B.66: Sample 19: Relevant Test Cases, part 2/2

B.20 Sample 20

```
Codiumate: Options | Test this function
2  ✓ def encode(message):
3  ✓     """
4     Assuming that all inputs are valid. This function takes a message, and encodes in such a
5     way that it swaps case of all letters, replaces all vowels in
6     the message with the letter that appears 2 places ahead of that
7     vowel in the english alphabet.
8     Assume only letters.
9
10    Examples:
11    >>> encode('test')
12    'TGST'
13    >>> encode('This is a message')
14    'tHKS KS C MGSSCGG'
15    """
16    vowels = "aeiouAEIOU"
17    vowels_replace = dict([(i, chr(ord(i) + 2)) for i in vowels])
18    message = message.swapcase()
19    return ''.join([vowels_replace[i] if i in vowels else i for i in message])
20
21  ✓ if __name__ == "__main__":
22    print(encode('!@#'))
23    print(encode('AEIOU'))
24    print(encode(' '))
25
```

Figure B.67: Sample 20: Function Under Test

```

1  # {#20} Test Case Selection Criteria for encode Function 93
2
3  ## Test Cases for `encode` Function
4
5  ### Boundary Values:
6  Test with boundary values where `message` is minimal or maximal.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1 | `""` | `""` |
11 | 2 | `"A"` | `"c"` |
12 | 3 | `"a"` | `"C"` |
13 | 4 | `"ZZZ"` | `"zzz"` |
14 | 5 | `"aeiou"` | `"CGKQW"` |
15 | 6 | `"AEIOU"` | `"cgkqw"` |
16
17 ### Typical Values:
18 Test with typical inputs where `message` contains various letters.
19
20 | Test Case | Input | Expected Output |
21 |-----|-----|-----|
22 | 7 | `"test"` | `"TGST"` |
23 | 8 | `"This is a message"` | `"tHKS KS C MGSSCGG"` |
24
25 ### Special Cases:
26 Consider special cases mentioned or implied in the docstring. For instance:
27 - Test with special characters
28
29 | Test Case | Input | Expected Output |
30 |-----|-----|-----|
31 | 9 | `"123"` | `"123"` |
32 | 9 | `"!@#"` | `"!@#"` |

```

Figure B.68: Sample 20: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  from experiment_sample_pool.function_93.prompt_93 import encode
5  # Generated by CodiumAI
6
7  import pytest
8
9  Codiumate: Add more tests
10 class TestEncode:
11     # Matches test case 7
12     # Should encode a message with only lowercase letters
13     def test_encode_lowercase(self):
14         assert encode('test') == 'TGST'
15         assert encode('this is a message') == 'tHKS KS C MGSSCGG'
16
17
18     # Matches test case 1
19     # Should encode an empty message
20     Codiumate: Add more tests
21     def test_encode_empty(self):
22         assert encode('') == ''
23
24     # Matches test case 4
25     # Should encode a message with only uppercase letters
26     Codiumate: Add more tests
27     def test_encode_uppercase(self):
28         assert encode('TEST') == 'TGST'
29         assert encode('THIS IS A MESSAGE') == 'tHKS KS C MGSSCGG'
30
31     # Matches test case 7
32     # Should encode a message with both lowercase and uppercase letters
33     Codiumate: Add more tests
34     def test_encode_lowercase(self):
35         assert encode('test') == 'TGST'
36         assert encode('this is a message') == 'tHKS KS C MGSSCGG'
37
38     # Matches test case 9
39     # Should encode a message with no vowels
40     Codiumate: Add more tests
41     def test_encode_no_vowels(self):
42         assert encode('xyz') == 'XYZ'
43         assert encode('123') == '123'
44         assert encode('!@#') == '!@#'
45         assert encode('') == ''
46         assert encode(' ') == ' '
47         assert encode('No vowels here') == 'nO vOwEls hErE'
48
49     # Matches test case 4, 5 ,6
50     # Should encode a message with only vowels
51     Codiumate: Add more tests
52     def test_encode_only_vowels(self):
53         assert encode('aaa') == 'CCC'
54         assert encode('AEIOU') == 'CGKQS'
55         assert encode('aeiou') == 'cgkqs'

```

Figure B.69: Sample 20: Relevant Test Cases, part 1/2

```

52 # Matches test case 1, 2
53 # Should encode a message with only one letter
    Codiumate: Add more tests
54 def test_encode_single_letter(self):
55     assert encode('a') == 'C'
56     assert encode('A') == 'C'
57     assert encode('b') == 'B'
58     assert encode('B') == 'B'
59     assert encode('z') == 'Z'
60     assert encode('Z') == 'Z'
61
62 # Matches test case 1
63 # Should encode a message with only one vowel
    Codiumate: Add more tests
64 def test_encode_single_vowel(self):
65     assert encode('a') == 'C'
66     assert encode('e') == 'G'
67     assert encode('i') == 'K'
68     assert encode('o') == 'Q'
69     assert encode('u') == 'W'
70
71 # Should encode a message with only one consonant
    Codiumate: Add more tests
72 def test_encode_single_consonant(self):
73     assert encode('b') == 'D'
74     assert encode('B') == 'd'
75
76 # Matches test case 7
77 # Should encode a message with all consonants
    Codiumate: Add more tests
78 def test_encode_consonants(self):
79     assert encode('test') == 'TGST'
80     assert encode('this is a message') == 'tHKS KS C MGSSCGG'
81
82 # Should encode a message with all vowels
    Codiumate: Add more tests
83 def test_encode_all_vowels(self):
84     assert encode('a') == 'C'
85     assert encode('e') == 'G'
86     assert encode('i') == 'K'
87     assert encode('o') == 'Q'
88     assert encode('u') == 'W'
89     assert encode('A') == 'C'
90     assert encode('E') == 'G'
91     assert encode('I') == 'K'
92     assert encode('O') == 'Q'
93     assert encode('U') == 'W'
94
95 # Matches test case 8
96 # Should encode a message with spaces between words
    Codiumate: Add more tests
97 def test_encode_with_spaces(self):
98     assert encode('This is a message') == 'tHKS KS C MGSSCGG'
99     assert encode('Hello World') == 'hJLLQ WQRLE'
100     assert encode('Python is fun') == 'pYTHON QS HXN'
101     assert encode('Testing encoding') == 'tQSTQNG QNCQDQNG'
102
103 # Should encode a message with spaces at the beginning and end
    Codiumate: Add more tests
104 def test_encode_message_with_spaces(self):
105     assert encode(' test ') == ' TGST '
106     assert encode(' This is a message ') == ' tHKS KS C MGSSCGG '

```

Figure B.70: Sample 20: Relevant Test Cases, part 2/2

B.21 Sample 21

```

Codiumate: Options | Test this function
2  def count_up_to(n):
3      """Assuming that all inputs are valid. This function takes a non-negative integer and returns an array of the first n
4      integers that are prime numbers and less than n.
5      for example:
6      count_up_to(5) => [2,3]
7      count_up_to(11) => [2,3,5,7]
8      count_up_to(0) => []
9      count_up_to(20) => [2,3,5,7,11,13,17,19]
10     count_up_to(1) => []
11     count_up_to(18) => [2,3,5,7,11,13,17]
12     """
13     primes = []
14     for i in range(2, n):
15         is_prime = True
16         for j in range(2, i):
17             if i % j == 0:
18                 is_prime = False
19                 break
20         if is_prime:
21             primes.append(i)
22     return primes

```

Figure B.71: Sample 21: Function Under Test

```

1  # {#21} Test Case Selection Criteria for count_up_to Function 96 You, 2 month
2
3  ## Test Cases for `count_up_to` Function
4
5  ### Boundary Values:
6  Test with boundary values where `n` is minimal or maximal.
7
8  | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1         | `0`  | `[ ]`           |
11 | 2         | `1`  | `[ ]`           |
12 | 3         | `2`  | `[ ]`           |
13 | 4         | `3`  | `[2]`          |
14 | 5         | `5`  | `[2, 3]`       |
15
16 ### Typical Values:
17 Test with typical inputs where `n` is a non-negative integer.
18
19 | Test Case | Input | Expected Output |
20 |-----|-----|-----|
21 | 6         | `11` | `[2, 3, 5, 7]`  |
22 | 6         | `18` | `[2, 3, 5, 7, 11, 13, 17]` |
23 | 6         | `20` | `[2, 3, 5, 7, 11, 13, 17, 19]` |
24
25 ### Special Cases:
26 Consider special cases mentioned or implied in the docstring. For instance:
27 - Test with a large number.
28
29 | Test Case | Input | Expected Output |
30 |-----|-----|-----|
31 | 7         | `100` | `[2, 3, 5, 7, ..., 97]` (primes) |
32 | 7         | `10000` | `[2, 3, 5, 7, ..., 9973]` (primes) |

```

Figure B.72: Sample 21: Selection Criteria

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 # Generated by CodiumAI
5 from experiment_sample_pool.function_96.prompt_96 import count_up_to
6
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestCountUpTo:
12     # Matches test case 1
13     # returns an empty list when input is 0
14     Codiumate: Add more tests
15     def test_empty_list_when_input_is_0(self):
16         assert count_up_to(0) == []
17
18     # Matches test case 2
19     # returns an empty list when input is 1
20     Codiumate: Add more tests
21     def test_empty_list_when_input_is_1(self):
22         assert count_up_to(1) == []
23
24     # Matches test case 5, 6, 1
25     # returns the correct list of prime numbers when input is a positive integer greater than 1
26     Codiumate: Add more tests
27     def test_returns_correct_list_of_prime_numbers(self):
28         assert count_up_to(5) == [2, 3]
29         assert count_up_to(11) == [2, 3, 5, 7]
30         assert count_up_to(20) == [2, 3, 5, 7, 11, 13, 17, 19]
31         assert count_up_to(1) == []
32         assert count_up_to(18) == [2, 3, 5, 7, 11, 13, 17]
```

Figure B.73: Sample 21: Relevant Test Cases

B.22 Sample 22

```
Codiumate: Options | Test this function
2 def closest_integer(value):
3     '''
4     Assuming that all inputs are valid.
5     This function takes a value (string) representing a number
6     and returns the closest integer to it. If the number is equidistant
7     from two integers, round it away from zero.
8
9     Examples
10    >>> closest_integer("10")
11        10
12    >>> closest_integer("15.3")
13        15
14
15    Note:
16    Rounding away from zero means that if the given number is equidistant
17    from two integers, the one you should return is the one that is the
18    farthest from zero. For example closest_integer("14.5") should
19    return 15 and closest_integer("-14.5") should return -15.
20    '''
21    from math import floor, ceil
22
23    if value.count('.') == 1:
24        # remove trailing zeros
25        while (value[-1] == '0'):
26            value = value[:-1]
27
28    num = float(value)
29    if value[-2:] == '.5':
30        if num > 0:
31            res = ceil(num)
32        else:
33            res = floor(num)
34    elif len(value) > 0:
35        res = int(round(num))
36    else:
37        res = 0
38
39    return res
```

Figure B.74: Sample 22: Function Under Test

```

1  ✓ # {#22} Test Case Selection Criteria for closest_integer Function 99      You,
2
3  ✓ ## Test Cases for `closest_integer` Function
4
5  ✓ ### Boundary Values:
6  Test with boundary values where `value` is minimal or maximal.
7
8  ✓ | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1 | `0` | `0` |
11 | 2 | `1` | `1` |
12 | 3 | `-1` | `-1` |
13 | 4 | `1.5` | `2` |
14 | 5 | `-1.5` | `-2` |
15 | 6 | `2147483647` | `2147483647` (max int) |
16 | 7 | `2147483647.5` | `2147483648` (max int) |
17 | 8 | `-2147483648.5` | `-2147483649` (min int)|
18
19 ✓ ### Typical Values:
20 Test with typical inputs where `value` contains various decimal numbers.
21
22 ✓ | Test Case | Input | Expected Output |
23 |-----|-----|-----|
24 | 2 | `10` | `10` |
25 | 3 | `-10` | `-10` |
26 | 4 | `15.3` | `15` |
27 | 5 | `14.9` | `15` |
28 | 6 | `-14.2` | `-14` |
29 | 7 | `-14.6` | `-15` |
30
31 ✓ ### Special Cases:
32 Consider special cases mentioned or implied in the docstring. For instance:
33 - Test with a number that is equidistant from two integers.
34
35 ✓ | Test Case | Input | Expected Output |
36 |-----|-----|-----|
37 | 8 | `14.5` | `15` |
38 | 9 | `-14.5` | `-15` |

```

Figure B.75: Sample 22: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 # Generated by CodiumAI
5 from experiment_sample_pool.function_99.prompt_99 import closest_integer
6
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestClosestInteger:
12     # Matches test case 2
13     # Returns the closest integer to a positive integer string.
14     def test_positive_integer_string(self):
15         assert closest_integer("10") == 10
16         assert closest_integer("100") == 100
17
18     # Matches test case 3
19     # Returns the closest integer to a negative integer string.
20     Codiumate: Add more tests
21     def test_negative_integer_string(self):
22         assert closest_integer("-10") == -10
23         assert closest_integer("-100") == -100
24
25     # Matches test case 2
26     # Returns the closest integer to a positive float string.
27     Codiumate: Add more tests
28     def test_positive_integer_string(self):
29         assert closest_integer("10") == 10
30         assert closest_integer("100") == 100
31
32     # Matches test case 8, 7, 6
33     # Returns the closest integer to a negative float string.
34     Codiumate: Add more tests
35     def test_negative_float_string(self):
36         assert closest_integer("-14.5") == -15
37         assert closest_integer("-3.8") == -4
38         assert closest_integer("-7.2") == -7
39
40     # Matches test case 8
41     # Returns the closest integer to a positive float string ending with '.5'.
42     def test_positive_float_string(self):
43         assert closest_integer("15.5") == 16
44         assert closest_integer("25.5") == 26
45         assert closest_integer("35.5") == 36

```

Figure B.76: Sample 22: Relevant Test Cases, part 1/2

```
45     # Matches test case 9
46     # Returns the closest integer to a negative float string ending with '.5'.
    Codiumate: Add more tests
47     def test_negative_float_ending_with_5(self):
48         assert closest_integer("-14.5") == -15
49         assert closest_integer("-9.5") == -10
50
51     # No match but a valid test case
52     # Returns the closest integer to a negative float string ending with '.0'.
    Codiumate: Add more tests
53     def test_negative_float_ending_with_zero(self):
54         assert closest_integer("-14.0") == -14
55         assert closest_integer("-15.0") == -15
56
57     # Matches test case 5
58     # Returns the closest integer to a positive float string with a decimal part greater than 0.5.
59     def test_positive_float_string(self):
60         assert closest_integer("15.6") == 16
61         assert closest_integer("20.9") == 21
62
63     # No match but a valid test case
64     # Returns the closest integer to a positive float string with a very large decimal part.
    Codiumate: Add more tests
65     def test_positive_float_string(self):
66         assert closest_integer("1234567890.1234567890") == 1234567890
67
68     # No match but a valid test case
69     # Returns the closest integer to a negative float string with a very large decimal part.
    Codiumate: Add more tests
70     def test_negative_float_string_with_large_decimal_part(self):
71         assert closest_integer("-1234567890.1234567890") == -1234567890
```

Figure B.77: Sample 22: Relevant Test Cases, part 2/2

B.23 Sample 23

```
Codiumate: Options | Test this function
2  def f(n):
3      """Assuming that all inputs are valid. This function takes n as a parameter,
4      and returns a list of size n, such that the value of the element at index i is the factorial of i if i is even
5      or the sum of numbers from 1 to i otherwise.
6      i starts from 1.
7      the factorial of i is the multiplication of the numbers from 1 to i (1 * 2 * ... * i).
8      Example:
9      f(5) == [1, 2, 6, 24, 15]
10     """
11     ret = []
12     for i in range(1,n+1):
13         if i%2 == 0:
14             x = 1
15             for j in range(1,i+1): x *= j
16             ret += [x]
17         else:
18             x = 0
19             for j in range(1,i+1): x += j
20             ret += [x]
21     return ret
```

Figure B.78: Sample 23: Function Under Test

```

1  ✓ # {#23} Test Case Selection Criteria for f Function      You, 2 months ago • s
2
3  ✓ ## Test Cases for `f` Function
4
5  ✓ ### Boundary Values:
6  Test with boundary values where `n` is minimal or maximal.
7
8  ✓ | Test Case | Input | Expected Output |
9  |-----|-----|-----|
10 | 1          | `0`  | `[ ]`           |
11 | 2          | `1`  | `[1]`           |
12 | 3          | `3`  | `[1, 2, 6]`     |
13 | 4          | `2`  | `[1, 2]`        |
14 | 4          | `4`  | `[1, 2, 6, 24]` |
15
16 ✓ ### Typical Values:
17 Test with typical inputs where `n` is a positive integer.
18
19 ✓ | Test Case | Input | Expected Output |
20 |-----|-----|-----|
21 | 3          | `5`  | `[1, 2, 6, 24, 15]` |
22 | 4          | `8`  | `[1, 2, 6, 24, 15, 720, 28, 40320]` |
23 | 4          | `10` | `[1, 2, 6, 24, 15, 720, 28, 40320, 45, 3628800]` |
24
25
26 ✓ ### Special Cases:
27 Consider special cases mentioned or implied in the docstring. For instance:
28 - Test with negative input.
29 - Test with a large number
30
31 ✓ | Test Case | Input | Expected Output |
32 |-----|-----|-----|
33 | 5          | `-5` | `[ ]`           |
34 | 6          | `100` | `[1, 2, 6, 24, ..., 933262154439441...000]` |
35

```

Figure B.79: Sample 23: Selection Criteria

```
1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  # Generated by CodiumAI
5  from experiment_sample_pool.function_106.prompt_106 import f
6
7
8  import pytest
9
10 Codiumate: Add more tests
11 class TestF:
12     # Matches test case 5: n is positive odd integer
13     # Returns a list of size n, where n is a positive integer.
14     Codiumate: Add more tests
15     def test_returns_list_of_size_n(self):
16         # Arrange
17         n = 5
18
19         # Act
20         result = f(n)
21
22         # Assert
23         assert len(result) == n
24
25     # Matches test case 1: n is 0
26     # Returns an empty list when n is 0.
27     Codiumate: Add more tests
28     def test_returns_empty_list_when_n_is_0(self):
29         # Arrange
30         n = 0
31
32         # Act
33         result = f(n)
34
35         # Assert
36         assert result == []
37
38     # Matches test case 2: n is 1
39     # Returns a list of size 1 when n is 1.
40     Codiumate: Add more tests
41     def test_returns_list_of_size_1(self):
42         # Arrange
43         n = 1
44
45         # Act
46         result = f(n)
47
48         # Assert
49         assert len(result) == n
```

Figure B.80: Sample 23: Relevant Test Cases

B.24 Sample 24

```
Codiumate: Options | Test this function
2  def check(s):
3      val = 0
4      for i in s:
5          if i == '(':
6              val = val + 1
7          else:
8              val = val - 1
9              if val < 0:
10                 return False
11         return True if val == 0 else False
12

Codiumate: Options | Test this function
13 def match_parens(lst):
14     """
15     Assuming that all inputs are valid. You are given a list of two strings, both strings consist of open
16     parentheses '(' or close parentheses ')' only.
17     Your job is to check if it is possible to concatenate the two strings in
18     some order, that the resulting string will be good.
19     A string S is considered to be good if and only if all parentheses in S
20     are balanced. For example: the string '(()())' is good, while the string
21     '())' is not.
22     Return 'Yes' if there's a way to make a good string, and return 'No' otherwise.
23
24     Examples:
25     match_parens(['()(', ')']) == 'Yes'
26     match_parens(['(', ')']) == 'No'
27     """
28
29     S1 = lst[0] + lst[1]
30     S2 = lst[1] + lst[0]
31     return 'Yes' if check(S1) or check(S2) else 'No'
```

Figure B.81: Sample 24: Function Under Test

```

1  # {#24} Test Case Selection Criteria for `match_parens` Function 109
2
3  ## Input Types:
4
5  ### Boundary Values:
6  Test with boundary values where `lst` contains minimal or maximal elements.
7
8  | Test Case | Input                | Expected Output |
9  |-----|-----|-----|
10 | 1         | `[', ']'`           | `Yes`          |
11 | 2         | `['(', ')']`        | `Yes`          |
12 | 3         | `['()', '()']`      | `Yes`          |
13 | 4         | `[')', '()']`       | `No`           |
14 | 4         | `['()', ')']`       | `No`           |
15
16 ### Typical Values:
17 Test with typical inputs where `lst` contains various combinations of parentheses.
18
19 | Test Case | Input                | Expected Output |
20 |-----|-----|-----|
21 | 5         | `['()(', ')']`      | `Yes`          |
22 | 6         | `['))', ')]']`      | `No`           |
23 | 7         | `['(((, ')))]']`    | `Yes`          |
24 | 8         | `['(((, '()())']`   | `No`           |
25 | 8         | `['()(', '())()']`  | `Yes`          |
26
27
28 ### Special Cases:
29 Consider special cases mentioned or implied in the docstring. For instance:
30 - Test with different string order.
31
32 | Test Case | Input                | Expected Output |
33 |-----|-----|-----|
34 | 9         | `[')', '(']`        | `Yes`          |
35 | 9         | `['(())', '(())(']` | `Yes`          |
36 | 9         | `[')(', ')(']`      | `No`           |
37 | 10        | `[')()(', '()(']`  | `No`           |
38 | 10        | `['))', '()']`     | `No`           |

```

Figure B.82: Sample 24: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 # Generated by CodiumAI
5 from experiment_sample_pool.function_119.prompt_119 import match_parens
6
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestMatchParens:
12     # Matches test case 5, 2
13     # The function returns 'Yes' when given two strings that can be concatenated in some order to form a good string.
14     Codiumate: Add more tests
15     def test_concatenated_strings(self):
16         assert match_parens(['()', '()']) == 'Yes'
17         assert match_parens(['(', ')']) == 'Yes'
18
19     # Matches test case 1
20     # The function returns 'No' when given two empty strings.
21     def test_empty_strings(self):
22         assert match_parens(['', '']) == 'No'
23         assert match_parens(['(', '']) == 'No'
24
25     # Matches test case 1
26     # The function returns 'Yes' when given two empty strings.
27     Codiumate: Add more tests
28     def test_empty_strings(self):
29         assert match_parens(['', '']) == 'Yes'
30
31     # Matches test case 6,
32     # The function returns 'No' when given two strings that cannot be concatenated in any order to form a good string.
33     Codiumate: Add more tests
34     def test_no_good_string_possible(self):
35         assert match_parens(['()', '()']) == 'No'
36         assert match_parens(['()', '()']) == 'No'
37         assert match_parens(['(', '()']) == 'No'
38
39     # Matches test case 5, 2
40     # The function returns 'Yes' when given two identical strings consisting of alternating open and close parentheses.
41     def test_identical_strings(self):
42         assert match_parens(['()()', '()']) == 'Yes'
43         assert match_parens(['(', '()']) == 'Yes'
44
45     # Matches test case 6
46     # The function returns 'No' when given two identical strings consisting of only close parentheses.
47     Codiumate: Add more tests
48     def test_identical_strings(self):
49         assert match_parens([')', '()']) == 'No'
50
51     # Matches test case 6
52     # The function returns 'No' when given two identical strings consisting of only open parentheses.
53     Codiumate: Add more tests
54     def test_no_identical_strings(self):
55         assert match_parens(['(', '()']) == 'No'
56         assert match_parens(['((((', '((((')] == 'No'
57         assert match_parens(['(((((', '(((((')] == 'No'
58
59     # Matches test case 5, 2
60     # The function returns 'Yes' when given two identical strings consisting of nested parentheses.
61     Codiumate: Add more tests
62     def test_identical_nested_parentheses(self):
63         assert match_parens(['()()', '()']) == 'Yes'
64         assert match_parens(['(', '()']) == 'Yes'
65
66
67

```

Figure B.83: Sample 24: Relevant Test Cases

B.25 Sample 25

```

Codiumate: Options | Test this function
2  def split_words(txt):
3      """
4          Assuming that all inputs are valid. Given a string of words, return a list of words split on whitespace, if no whitespaces exists in the text you
5          should split on commas ','. If no commas exists you should return the number of lower-case letters with odd order in the
6          alphabet, ord('a') = 0, ord('b') = 1, ... ord('z') = 25
7          Examples
8          split_words("Hello world!") → ["Hello", "world!"]
9          split_words("Hello,world!") → ["Hello", "world!"]
10         split_words("abcdef") == 3
11         """
12         if " " in txt:
13             return txt.split()
14         elif "," in txt:
15             return txt.replace(' ', ',').split()
16         else:
17             return len([i for i in txt if i.islower() and ord(i)%2 == 0])
18

```

Figure B.84: Sample 25: Function Under Test

```

1  # {#25} Test Case Selection Criteria for `split_words` Function      You, 2 months ago • select
2
3  ## Test Cases for `split_words` Function. [^1]
4
5  ### Boundary Values:
6  Test with boundary values where `txt` is minimal or maximal.
7
8  | Test Case | Input                | Expected Output |
9  |-----|-----|-----|
10 | 1         | ``                 | `0`             |
11 | 2         | `a`               | `0`             |
12 | 3         | `Hello world!`    | `["Hello", "world!"]` |
13 | 4         | `Hello,world!`    | `["Hello", "world!"]` |
14 | 5         | `abcdefghijklmnopqrsuvwxyz` | `13`           |
15
16 ### Typical Values:
17 Test with typical inputs where `txt` contains various combinations of words and punctuation.
18
19 | Test Case | Input                | Expected Output |
20 |-----|-----|-----|
21 | 6         | `Hello, My name is Eva` | `['Hello,', 'My', 'name', 'is', 'Eva']` |
22 | 6         | `How are you?`      | `["How", "are", "you?"]` |
23
24
25 ### Special Cases:
26 Consider special cases mentioned or implied in the docstring. For instance:
27 - Test with mixed of space and comma.
28 - Test with multiple commas.
29
30 | Test Case | Input                | Expected Output |
31 |-----|-----|-----|
32 | 7         | `Python,is awesome` | `["Python,is", "awesome"]` |
33 | 8         | `Python is,awesome` | `["Python", "is,", "awesome"]` |
34 | 8         | `Python,is,awesome` | `["Python", "is", "awesome"]` |
35

```

Figure B.85: Sample 25: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 # Generated by CodiumAI
5 from experiment_sample_pool.function_125.prompt_125 import split_words
6
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestSplitWords:
12     # Matches test case 3, 4, 5
13     # Should split a string of words on whitespace
14     def test_split_on_whitespace(self):
15         assert split_words("Hello world!") == ["Hello", "world!"]
16         assert split_words("Hello,world!") == ["Hello", "world!"]
17         assert split_words("abcdef") == 3
18
19     # No matching test case but valid
20     # Should handle input with only whitespaces
21     Codiumate: Add more tests
22     def test_handle_only_whitespaces(self):
23         assert split_words(" ") == []
24         assert split_words(" , , ") == []
25         assert split_words("a,b,c,d,e,f") == 3
26
27     # Matches test case 4, 4, 5
28     # Should split a string of words on commas
29     Codiumate: Add more tests
30     def test_split_on_commas(self):
31         assert split_words("Hello,world!") == ["Hello", "world!"]
32         assert split_words("Hello,there!") == ["Hello", "there!"]
33         assert split_words("abcdef") == 3
34
35     # Matches test case 3, 3, 5
36     # Should return the number of lowercase letters with odd order in the alphabet if no whitespaces or commas exist
37     def test_split_on_whitespace(self):
38         assert split_words("Hello world!") == ["Hello", "world!"]
39         assert split_words("Hello,world!") == ["Hello", "world!"]
40         assert split_words("abcdef") == 3
41
42     # Matches test case 1
43     # Should handle empty string input
44     Codiumate: Add more tests
45     def test_empty_string_input(self):
46         assert split_words("") == 0
47
48     # Matches test case 3, 4, 5
49     # Should handle input with only one word
50     def test_split_on_whitespace(self):
51         assert split_words("Hello world!") == ["Hello", "world!"]
52         assert split_words("Hello,world!") == ["Hello", "world!"]
53         assert split_words("abcdef") == 3

```

Figure B.86: Sample 25: Relevant Test Cases, part 1/3

```

52     # Matches test case 2
53     # Should handle input with only one letter
    Codiumate: Add more tests
54     def test_handle_input_with_one_letter(self):
55         assert split_words("a") == 0
56         assert split_words("b") == 1
57         assert split_words("c") == 0
58         assert split_words("d") == 1
59         assert split_words("e") == 0
60         assert split_words("f") == 1
61         assert split_words("g") == 0
62         assert split_words("h") == 1
63         assert split_words("i") == 0
64         assert split_words("j") == 1
65         assert split_words("k") == 0
66         assert split_words("l") == 1
67         assert split_words("m") == 0
68         assert split_words("n") == 1
69         assert split_words("o") == 0
70         assert split_words("p") == 1
71         assert split_words("q") == 0
72         assert split_words("r") == 1
73         assert split_words("s") == 0
74         assert split_words("t") == 1
75         assert split_words("u") == 0
76         assert split_words("v") == 1
77         assert split_words("w") == 0
78         assert split_words("x") == 1
79         assert split_words("y") == 0
80         assert split_words("z") == 1
81
82     # No matching test case but valid
83     # Should handle input with only one uppercase letter
    Codiumate: Add more tests
84     def test_handle_input_with_one_uppercase_letter(self):
85         assert split_words("A") == 0
86
87     # Matches test case 2
88     # Should handle input with only one lowercase letter
    Codiumate: Add more tests
89     def test_handle_input_with_one_lowercase_letter(self):
90         assert split_words("a") == 0
91         assert split_words("b") == 1
92         assert split_words("c") == 0
93         assert split_words("d") == 1
94         assert split_words("e") == 0
95         assert split_words("f") == 1
96         assert split_words("g") == 0
97         assert split_words("h") == 1
98         assert split_words("i") == 0
99         assert split_words("j") == 1
100        assert split_words("k") == 0
101        assert split_words("l") == 1
102        assert split_words("m") == 0
103        assert split_words("n") == 1
104        assert split_words("o") == 0
105        assert split_words("p") == 1
106        assert split_words("q") == 0
107        assert split_words("r") == 1
108        assert split_words("s") == 0
109        assert split_words("t") == 1
110        assert split_words("u") == 0
111        assert split_words("v") == 1
112        assert split_words("w") == 0
113        assert split_words("x") == 1
114        assert split_words("y") == 0
115        assert split_words("z") == 1

```

Figure B.87: Sample 25: Relevant Test Cases, part 2/3

```

116
117
118     # Matches test case 4, 5
119     # Should handle input with only commas
    Codiumate: Add more tests
120     def test_handle_input_with_only_commas(self):
121         assert split_words("Hello,world!") == ["Hello", "world!"]
122         assert split_words("abcdef") == 3
123
124     # No matching test case but valid if return == 0
125     # Should handle input with only uppercase letters
    Codiumate: Add more tests
126     def test_handle_input_with_only_uppercase_letters(self):
127         assert split_words("HELLO") == 4
128         assert split_words("WORLD") == 5
129         assert split_words("PYTHON") == 6
130
131     # Matches test case 5
132     # Should handle input with only even numbered lowercase letters
    Codiumate: Add more tests
133     def test_handle_input_with_only_even_numbered_lowercase_letters(self):
134         assert split_words("abcdef") == 3
135
136     # Duplicate test case
137     # Should handle input with only odd numbered lowercase letters
    Codiumate: Add more tests
138     def test_odd_lowercase_letters(self):
139         assert split_words("abcdef") == 3
140
141     # No mathcing test case but valid
142     # Should handle input with leading/trailing whitespaces
    Codiumate: Add more tests
143     def test_leading_trailing_whitespaces(self):
144         assert split_words(" Hello world! ") == ["Hello", "world!"]
145         assert split_words(" Hello,world! ") == ["Hello", "world!"]
146         assert split_words(" abcdef ") == 3
147
148     # Duplicate test cases
149     # Should handle input with mixed letters, numbers, and special characters
    Codiumate: Add more tests
150     def test_split_on_mixed_input(self):
151         assert split_words("Hello world!") == ["Hello", "world!"]
152         assert split_words("Hello,world!") == ["Hello", "world!"]
153         assert split_words("abcdef") == 3
154
155     # Duplicate test cases
156     # Should handle input with mixed whitespaces and commas
    Codiumate: Add more tests
157     def test_split_on_whitespace(self):
158         assert split_words("Hello world!") == ["Hello", "world!"]
159         assert split_words("Hello,world!") == ["Hello", "world!"]
160         assert split_words("abcdef") == 3

```

Figure B.88: Sample 25: Relevant Test Cases, part 3/3

B.26 Sample 26

```

Codiumate: Options | Test this function
2  def is_equal_to_sum_even(n):
3      """Assuming that all inputs are valid.
4      Evaluate whether the given number n can be written as the sum of exactly 4 positive even numbers
5      Example
6      is_equal_to_sum_even(4) == False
7      is_equal_to_sum_even(6) == False
8      is_equal_to_sum_even(8) == True
9      """
10     return n%2 == 0 and n >= 8
11

```

Figure B.89: Sample 26: Function Under Test

```

1  # {#26} Test Case Selection Criteria for `is_equal_to_sum_even` Function 138
2
3
4  ### Boundary Values:
5  Test with boundary values where `n` is minimal or maximal.
6
7  | Test Case | Input | Expected Output |
8  |-----|-----|-----|
9  | 1         | `0`   | `False`         |
10 | 2         | `3`   | `False`         |
11 | 3         | `2`   | `False`         |
12 | 4         | `8`   | `True`          |
13 | 5         | `9999998` | `True`          |
14 | 5         | `1000000` | `True`          |
15 | 6         | `9999999` | `False`         |
16
17 ### Typical Values:
18 Test with typical inputs where `n` contains various even numbers.
19
20 | Test Case | Input | Expected Output |
21 |-----|-----|-----|
22 | 7         | `6`   | `False`         |
23 | 7         | `12`  | `False`         |
24 | 7         | `28`  | `False`         |
25 | 8         | `16`  | `True`          |
26 | 8         | `24`  | `True`          |
27 | 8         | `32`  | `True`          |
28 | 8         | `1000` | `True`          |
29

```

Figure B.90: Sample 26: Selection Criteria

```
1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  # Generated by CodiumAI
5  from experiment_sample_pool.function_138.prompt_138 import is_equal_to_sum_even
6
7
8  import pytest
9
10 Codiumate: Add more tests
11 class TestIsEqualToSumEven:
12     # Match test case 4: minimal true,
13     # Match test case 7: typical false
14     # The function returns True when given an even number greater than or equal to 8.
15     def test_even_number_greater_than_or_equal_to_8(self):
16         assert is_equal_to_sum_even(8) == True
17         assert is_equal_to_sum_even(10) == True
18
19     # Match test case 1: minimal false
20     # The function returns False when given 0.
21     Codiumate: Add more tests
22     def test_zero(self):
23         assert is_equal_to_sum_even(0) == False
24
25     # Match test case 2: minimal odd int false
26     # The function returns False when given an odd number.
27     Codiumate: Add more tests
28     def test_odd_number(self):
29         assert is_equal_to_sum_even(3) == False
30         assert is_equal_to_sum_even(5) == False
31
32     # Match test case 7: typical false
33     # The function returns False when given an even number less than 8.
34     Codiumate: Add more tests
35     def test_even_number_less_than_8(self):
36         assert is_equal_to_sum_even(4) == False
37         assert is_equal_to_sum_even(6) == False
```

Figure B.91: Sample 26: Relevant Test Cases, part 1/3

```
36 # Match test case 3: minimal even int false
37 # The function returns False when given 2.
    Codiumate: Add more tests
38 ✓ def test_returns_false_when_given_2(self):|
39     |     assert is_equal_to_sum_even(2) == False
40
41 # Match test case 7: typical false
42 # The function returns False when given 6.
    Codiumate: Add more tests
43 ✓ def test_returns_false_when_given_6(self):
44     |     assert is_equal_to_sum_even(6) == False
45
46 # Match test case 4: minimal even int true
47 # Match test case 7: typical true
48 # The function returns True when given 8.
49 ✓ def test_even_number_greater_than_or_equal_to_8(self):
50     |     assert is_equal_to_sum_even(8) == True
51     |     assert is_equal_to_sum_even(10) == True
52
53 # Duplicate
54 # The function should return a boolean value.
    Codiumate: Add more tests
55 ✓ def test_even_number_greater_than_or_equal_to_8(self):
56     |     assert is_equal_to_sum_even(8) == True
57     |     assert is_equal_to_sum_even(10) == True
58
59 # Match test case 5: maximal true
60 # Match test case 6: maximal false
61 # The function should be able to handle large input values.
    Codiumate: Add more tests
62 ✓ def test_large_input_values(self):
63     |     assert is_equal_to_sum_even(1000000) == True
64     |     assert is_equal_to_sum_even(9999998) == True
65     |     assert is_equal_to_sum_even(9999999) == False
```

Figure B.92: Sample 26: Relevant Test Cases, part 2/3

B.27 Sample 27

```

1  def digits_sum(n):          You, 2 months ago * wrote selection criteria for function 26-30
2      neg = 1
3      if n < 0: n, neg = -1 * n, -1
4      n = [int(i) for i in str(n)]
5      n[0] = n[0] * neg
6      return sum(n)
7
Codiumate: Options | Test this function
8  def order_by_points(nums):
9      """
10     Assuming that all inputs are valid. This function sorts the given list of integers
11     in ascending order according to the sum of their digits.
12     Note: if there are several items with similar sum of their digits,
13     order them based on their index in original list.
14
15     For example:
16     >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
17     >>> order_by_points([]) == []
18     """
19     return sorted(nums, key=digits_sum)
20

```

Figure B.93: Sample 27: Function Under Test

```

1  # {#27} Test Case Selection Criteria for Function 138
2
3  ## Test Cases for `order_by_points` Function
4
5  ### Boundary Values:
6  Test with boundary values where the input list is empty or contains only one element.
7  These scenarios should cover the cases where:          You, 1 second ago * Uncommitted changes
8  - The input list is empty.
9  - The input list contains a single element.
10 - Test positive list.
11 - Test negative list.
12 - Test mixed list of pos and neg.
13
14 | Test Case | Input          | Expected Output |
15 |-----|-----|-----|
16 | 1         | `[ ]`         | `[ ]`           |
17 | 2         | `[0]`        | `[0]`           |
18 | 3         | `[5]`        | `[5]`           |
19 | 4         | `[7, 3, 1]`  | `[1, 3, 7]`     |
20 | 5         | `[-30, -3, -61]` | `[-61, -30, -3]` |
21
22 ### Typical Values:
23 Test with typical lists of integers that may occur in a real-world scenario.
24 These could include lists with different lengths and containing various integers.
25
26 | Test Case | Input          | Expected Output |
27 |-----|-----|-----|
28 | 6         | `[1, 11, -1, -11, -12]` | `[-1, -11, 1, -12, 11]` |
29 | 6         | `[1, -11, -32, 43, 54, -98, 2, -3]` | `[-3, -32, -98, -11, 1, 2, 43, 54]` |
30 | 6         | `[0,6,6,-76,-21,23,4]` | `[-76, -21, 0, 4, 23, 6, 6]` |
31 | 6         | `[-11, -111, -1111, 0, -1]` | `[-1, -11, 0, -111, -1111]` |
32 | 6         | `[1,2,3,4,5,6,7,8,9,10,11]` | `[1, 10, 2, 11, 3, 4, 5, 6, 7, 8, 9]` |
33

```

Figure B.94: Sample 27: Selection Criteria

```

1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  from experiment_sample_pool.function_145.prompt_145 import order_by_points
5  # Generated by CodiumAI
6
7  import pytest
8
9  Codiumate: Add more tests
10 class TestOrderByPoints:
11     # Match test case 4: sort positive integers
12     # Sorts a list of positive integers in ascending order based on the sum of their digits.
13     def test_sort_positive_integers(self):
14         assert order_by_points([123, 456, 789]) == [123, 456, 789]
15         assert order_by_points([789, 456, 123]) == [123, 456, 789]
16
17     # Match test case 3: sort a single element
18     # Sorts a list of integers with a single element.
19     Codiumate: Add more tests
20     def test_sort_single_element(self):
21         assert order_by_points([5]) == [5]
22         assert order_by_points([-10]) == [-10]
23
24     # Match test case 6: sort mixed integers
25     # Match test case 1: sort an empty list
26     # Sorts a list of mixed positive and negative integers in ascending order based on the sum of their digits.
27     Codiumate: Add more tests
28     def test_sort_mixed_integers(self):
29         assert order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
30         assert order_by_points([10, -10, 100, -100]) == [-100, -10, 10, 100]
31         assert order_by_points([0, -5, 5]) == [0, -5, 5]
32         assert order_by_points([-9, -99, 9, 99]) == [-9, -99, 9, 99]
33         assert order_by_points([]) == []
34
35     # Match test case 4: sort positive integers
36     # Sorts a list of integers with the same digit sum in ascending order based on their index in the original list.
37     def test_sort_positive_integers(self):
38         assert order_by_points([123, 456, 789]) == [123, 456, 789]
39         assert order_by_points([789, 456, 123]) == [123, 456, 789]
40
41     # Match test case 1: sort an empty list
42     # Returns an empty list when given an empty list.
43     Codiumate: Add more tests
44     def test_empty_list_input(self):
45         assert order_by_points([]) == []
46
47     # Match test case 4: sort positive integers
48     # Sorts a list of integers with two elements.
49     Codiumate: Add more tests
50     def test_sort_two_elements(self):
51         assert order_by_points([5, 2]) == [2, 5]

```

Figure B.95: Sample 27: Relevant Test Cases, part 1/3

B.28 Sample 28

```
Codiumate: Options | Test this function
2  def bf(planet1, planet2):
3      """
4      There are eight planets in our solar system: the closest to the Sun
5      is Mercury, the next one is Venus, then Earth, Mars, Jupiter, Saturn,
6      Uranus, Neptune.
7      This function takes two planet names as strings planet1 and planet2.
8      The function should return a tuple containing all planets whose orbits are
9      located between the orbit of planet1 and the orbit of planet2, sorted by
10     the proximity to the sun.
11     The function should return an empty tuple if planet1 or planet2
12     are not correct planet names.
13     Examples
14     bf("Jupiter", "Neptune") ==> ("Saturn", "Uranus")
15     bf("Earth", "Mercury") ==> ("Venus")
16     bf("Mercury", "Uranus") ==> ("Venus", "Earth", "Mars", "Jupiter", "Saturn")
17     """
18     planet_names = ("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")
19     if planet1 not in planet_names or planet2 not in planet_names or planet1 == planet2:
20         return ()
21     planet1_index = planet_names.index(planet1)
22     planet2_index = planet_names.index(planet2)
23     if planet1_index < planet2_index:
24         return (planet_names[planet1_index + 1 : planet2_index])
25     else:
26         return (planet_names[planet2_index + 1 : planet1_index])
27
```

Figure B.97: Sample 28: Function Under Test

```

1  # {#28} Test Case Selection Criteria for `bf` Function 148      You, 2 months ago • wrote selection cri
2
3  ### Boundary Values:
4  Test with boundary values where `planet1` and `planet2` are minimal or maximal.
5
6  | Test Case | Input                                | Expected Output |
7  |-----|-----|-----|
8  | 1         | ` "Mars", ` "Jupiter" ` | ` () `         |
9  | 2         | ` "Mars", ` "Saturn" ` | ` ("Jupiter") ` |
10 | 3         | ` "Mercury", ` "Neptune" ` | ` ("Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus") ` |
11
12 ### Typical Values:
13 Test with typical inputs where `planet1` and `planet2` contain various planet names.
14
15 | Test Case | Input                                | Expected Output |
16 |-----|-----|-----|
17 | 4         | ` "Jupiter", ` "Neptune" ` | ` ("Saturn", "Uranus") ` |
18 | 4         | ` "Mercury", ` "Mars" ` | ` ("Venus", "Earth") ` |
19 | 4         | ` "Mars", ` "Neptune" ` | ` ("Jupiter", "Saturn", "Uranus") ` |
20
21
22 ### Special Cases:
23 Consider special cases mentioned or implied in the docstring. For instance:
24 - Test with cases where `planet1` or `planet2` are not correct planet names.
25 - Test with planet names in reverse order.
26 - Test with same planet names.
27
28 | Test Case | Input                                | Expected Output |
29 |-----|-----|-----|
30 | 5         | ` "Mars", ` "Pluto" ` | ` () `         |
31 | 5         | ` "Sun", ` "Earth" ` | ` () `         |
32 | 6         | ` "Earth", ` "Mercury" ` | ` ("Venus") ` |
33 | 7         | ` "Neptune", ` "Mercury" ` | ` ("Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus") ` |
34 | 8         | ` "Mars", ` "Earth" ` | ` () `         |
35 | 9         | ` "Mars", ` "Mars" ` | ` () `         |
36

```

Figure B.98: Sample 28: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 # Generated by CodiumAI
5 from experiment_sample_pool.function_148.prompt_148 import bf
6
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestBf:
12
13     # Match test case 4: Returns a tuple of planets in the correct order, sorted by proximity to the sun.
14     # Match test case 6: Reverse input planets. Returns a tuple of planets in the correct order, sorted by proximity to the sun.
15     # Returns a tuple of planets between two valid planet names.
16     def test_valid_planets(self):
17         assert bf("Jupiter", "Neptune") == ("Saturn", "Uranus")
18         assert bf("Earth", "Mercury") == ("Venus")
19
20     # Match test case 1: Returns an empty tuple if planet1 or planet2 are next to each other.
21     # Returns an empty tuple if planet1 and planet2 are adjacent planets.
22     Codiumate: Add more tests
23     def test_adjacent_planets(self):
24         assert bf("Mercury", "Venus") == ()
25         assert bf("Earth", "Mars") == ()
26
27     # Match test case 5: Returns an empty tuple if planet1 and planet2 are not valid planet names.
28     # Match test case 9: Returns an empty tuple if planet1 or planet2 are the same names
29     # Returns an empty tuple if planet1 or planet2 are not valid planet names.
30     Codiumate: Add more tests
31     def test_invalid_planets(self):
32         assert bf("Pluto", "Mars") == ()
33         assert bf("Earth", "Pluto") == ()
34         assert bf("Pluto", "Pluto") == ()
35         assert bf("Mars", "Mars") == ()
36
37     # Match test case 9: Returns an empty tuple if planet1 or planet2 are the same names
38     # Returns an empty tuple if planet1 and planet2 are the same planet.
39     Codiumate: Add more tests
40     def test_same_planet(self):
41         assert bf("Mercury", "Mercury") == ()
42         assert bf("Venus", "Venus") == ()
43         assert bf("Earth", "Earth") == ()
44         assert bf("Mars", "Mars") == ()
45         assert bf("Jupiter", "Jupiter") == ()
46         assert bf("Saturn", "Saturn") == ()
47         assert bf("Uranus", "Uranus") == ()
48         assert bf("Neptune", "Neptune") == ()
49
50     # Match test case 4: Returns a tuple of planets in the correct order, sorted by proximity to the sun.
51     # Match test case 6: Reverse input planets. Returns a tuple of planets in the correct order, sorted by proximity to the sun.
52     # Returns a tuple of planets in the correct order, sorted by proximity to the sun.
53     Codiumate: Add more tests
54     def test_valid_planets(self):
55         assert bf("Jupiter", "Neptune") == ("Saturn", "Uranus")
56         assert bf("Earth", "Mercury") == ("Venus")

```

Figure B.99: Sample 28: Relevant Test Cases

B.29 Sample 29

```
Codiumate: Options | Test this function
2  def x_or_y(n, x, y):
3      """Assuming that all inputs are valid.
4      A simple program which should return the value of x if n is
5      a prime number and should return the value of y otherwise.
6
7      Examples:
8      for x_or_y(7, 34, 12) == 34
9      for x_or_y(15, 8, 5) == 5
10
11     """
12     if n == 1:
13         return y
14     for i in range(2, n):
15         if n % i == 0:
16             return y
17             break
18     else:
19         return x
```

Figure B.100: Sample 29: Function Under Test

```

1  # {#29} Test Case Selection Criteria for `x_or_y` Function 150
2
3  ## Test Cases for `x_or_y` Function
4
5  ### Boundary Values:
6  Test with boundary values where `n` is minimal or special values like 0 and 1.
7
8  | Test Case | Input | x | y | Expected Output |
9  |-----|-----|---|---|-----|
10 | 1         | `0`  | `5` | `3` | `3`
11 | 2         | `1`  | `5` | `3` | `3`
12 | 3         | `2`  | `5` | `3` | `5`
13 | 4         | `4`  | `5` | `3` | `3`
14
15 ### Typical Values:
16 Test with typical inputs where `n` is a prime or non-prime number.
17
18 | Test Case | Input | x | y | Expected Output |
19 |-----|-----|---|---|-----|
20 | 3         | `7`  | `34` | `12` | `34`
21 | 3         | `31` | `15` | `20` | `15`
22 | 3         | `1259` | `3` | `52` | `3`
23 | 3         | `7919` | `-1` | `12` | `-1`
24 | 4         | `15` | `8` | `5` | `5`
25 | 4         | `20` | `6` | `9` | `6`
26
27 ### Special Cases:
28 Consider special cases mentioned or implied in the docstring. For instance:
29 - Test with negative numbers.
30
31 | Test Case | Input | x | y | Expected Output |
32 |-----|-----|---|---|-----|
33 | 5         | `-5` | `9` | `3` | `3`
34 | 5         | `-10` | `2` | `4` | `4`
35
36

```

Figure B.101: Sample 29: Selection Criteria

```
1  import sys
2  import os
3  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4  # Generated by CodiumAI
5  from experiment_sample_pool.function_150.prompt_150 import x_or_y
6
7
8  import pytest
9
10  Codiumate: Add more tests
11  class TestXOrY:
12      # Match test case 3: Returns x if n is a prime number
13      # Returns x if n is a prime number
14      Codiumate: Add more tests
15      def test_returns_x_if_n_is_prime(self):
16          assert x_or_y(7, 34, 12) == 34
17          assert x_or_y(2, 8, 5) == 8
18
19      # Match test case 2: Returns y if n is 0
20      # Returns y if n is 0
21      Codiumate: Add more tests
22      def test_returns_y_if_n_is_zero(self):
23          assert x_or_y(0, 34, 12) == 12
24          assert x_or_y(0, 8, 5) == 5
25
26      # Match test case 4: Returns y if n is not a prime number
27      # Returns y if n is not a prime number
28      Codiumate: Add more tests
29      def test_returns_y_if_n_is_not_prime(self):
30          assert x_or_y(15, 8, 5) == 5
31          assert x_or_y(9, 34, 12) == 12
32
33      # Match test case 1: Returns y if n is 1
34      # Returns y if n is 1
35      Codiumate: Add more tests
36      def test_returns_y_if_n_is_1(self):
37          assert x_or_y(1, 34, 12) == 12
38
39      # Match test case 5: Returns y if n is negative
40      # Returns y if n is negative
41      Codiumate: Add more tests
42      def test_returns_y_if_n_is_negative(self):
43          assert x_or_y(-5, 34, 12) == 12
44          assert x_or_y(-10, 8, 5) == 5
45
46  40
```

Figure B.102: Sample 29: Relevant Test Cases

B.30 Sample 30

```

Codiumate: Options | Test this function
2  def compare(game,guess):
3      """Assuming that all inputs are valid.
4          I think we all remember that feeling when the result of some long-awaited
5          event is finally known. The feelings and thoughts you have at that moment are
6          definitely worth noting down and comparing.
7          Your task is to determine if a person correctly guessed the results of a number of matches.
8          You are given two arrays of scores and guesses of equal length, where each index shows a match.
9          Return an array of the same length denoting how far off each guess was. If they have guessed correctly,
10         the value is 0, and if not, the value is the absolute difference between the guess and the score.
11
12
13         example:
14
15         compare([1,2,3,4,5,1],[1,2,3,4,2,-2]) -> [0,0,0,0,3,3]
16         compare([0,5,0,0,0,4],[4,1,1,0,0,-2]) -> [4,4,1,0,0,6]
17         """
18         return [abs(x-y) for x,y in zip(game,guess)]
19

```

Figure B.103: Sample 30: Function Under Test

```

1  # {#30} Test Case Selection Criteria for `compare` Function 152  You, 2 months ago * wrote selection criteria
2
3  ### Boundary Values:
4  Test with boundary values where `game scores` and `guesses` contain minimal or maximal elements.
5  - Test with empty lists.
6  - Test single elements with zeros.
7  - Test single element with integer.
8  - Test identical arrays
9  - Test single element with large positive int
10
11 | Test Case | Input | Expected Output |
12 |-----|-----|-----|
13 | 1 | `[]`, `[]` | `[]` |
14 | 2 | `[0]`, `[0]` | `[0]` |
15 | 3 | `[2]`, `[1]` | `[1]` |
16 | 4 | `[1, 2, 3, 4, 5, 6]`, `[1, 2, 3, 4, 5, 6]` | `[0, 0, 0, 0, 0, 0]` |
17 | 5 | `[1000000]`, `[999999]` | `[1]` |
18
19 ### Typical Values:
20 Test with typical inputs where `game scores` and `guesses` contain various combinations of scores and guesses.
21 - Test with some match elements
22 - Test with no match element
23 - Test with combination of positives and negatives in game scores or guesses
24
25 | Test Case | Input | Expected Output |
26 |-----|-----|-----|
27 | 6 | `[1, 2, 3, 4, 5, 1]`, `[1, 2, 3, 4, 2, -2]` | `[0, 0, 0, 0, 3, 3]` |
28 | 6 | `[10, 15, 20]`, `[9, 16, 25]` | `[1, 1, 5]` |
29 | 6 | `[0, 5, 0, 0, 0, 4]`, `[4, 1, 1, 0, 0, -2]` | `[4, 4, 1, 0, 0, 6]` |
30 | 7 | `[3, 3, 3, -3, -3]`, `[0, 0, 0, 0, 0]` | `[3, 3, 3, 3, 3]` |
31

```

Figure B.104: Sample 30: Selection Criteria

```

1 import sys
2 import os
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
4 # Generated by CodiumAI
5 from experiment_sample_pool.function_152.prompt_152 import compare
6
7
8 import pytest
9
10 Codiumate: Add more tests
11 class TestCompare:
12     # Match test case 1: empty arrays
13     # Returns an empty array when input arrays are empty
14     Codiumate: Add more tests
15     def test_empty_arrays(self):
16         assert compare([], []) == []
17
18     # Match test case 3: identical arrays
19     # Returns an array with 0 values when input arrays are identical
20     Codiumate: Add more tests
21     def test_identical_arrays(self):
22         assert compare([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]) == [0, 0, 0, 0, 0]
23         assert compare([0, 0, 0], [0, 0, 0]) == [0, 0, 0]
24         assert compare([-1, -2, -3], [-1, -2, -3]) == [0, 0, 0]
25
26     # Match test case 6: arrays of different integer values
27     # Works correctly with input arrays of negative values
28     Codiumate: Add more tests
29     def test_negative_values(self):
30         assert compare([-1,-2,-3,-4,-5,-1],[-1,-2,-3,-4,-2,2]) == [0,0,0,0,3,3]
31
32     # Match test case 6: arrays of different integer values
33     # Works correctly with input arrays of large integer values
34     Codiumate: Add more tests
35     def test_large_integer_values(self):
36         assert compare([1000000000, 2000000000, 3000000000], [1000000000, 2000000000, 3000000000]) == [0, 0, 0]
37         assert compare([1000000000, 2000000000, 3000000000], [999999999, 1999999999, 2999999999]) == [1, 1, 1]
38         assert compare([1000000000, 2000000000, 3000000000], [1000000001, 2000000001, 3000000001]) == [1, 1, 1]
39         assert compare([1000000000, 2000000000, 3000000000], [500000000, 1500000000, 2500000000]) == [500000000, 500000000, 500000000] # false positive
40
41     # Match test case 6: arrays of different integer values
42     # Works correctly with input arrays of mixed integer values
43     Codiumate: Add more tests
44     def test_compare_with_mixed_integer_values(self):
45         assert compare([1,2,3,4,5,1],[1,2,3,4,2,-2]) == [0,0,0,0,3,3]
46         assert compare([0,5,0,0,0,4],[4,1,1,0,0,-2]) == [4,4,1,0,0,6]
47
48     # Works correctly with input arrays of maximum integer values
49     Codiumate: Add more tests
50     def test_maximum_integer_values(self):
51         assert compare([9999999999999999999], [9999999999999999999]) == [0]

```

Figure B.105: Sample 30: Relevant Test Cases, part 1/2

```

47 # Match test case 2: zero values
48 # Match test case 3: arrays of single integer values
49 # Works correctly with input arrays of minimum integer values
    Codiumate: Add more tests
50 def test_minimum_integer_values(self):
51     assert compare([0], [0]) == [0]
52     assert compare([-1], [-1]) == [0]
53     assert compare([-100], [-100]) == [0]
54     assert compare([0], [1]) == [1]
55     assert compare([-1], [1]) == [2]
56     assert compare([-100], [100]) == [200]
57
58
59 # Match test case 6: Test with some match elements
60 # Returns an array with correct values when input arrays are of length 5
    Codiumate: Add more tests
61 def test_correct_values_length_5(self):
62     assert compare([1, 2, 3, 4, 5], [1, 2, 3, 4, 2]) == [0, 0, 0, 0, 3]
63
64 # Match test case 4: Test identical arrays
65 # Returns an array with correct values when input arrays are of length 4
    Codiumate: Add more tests
66 def test_correct_values_length_4(self):
67     assert compare([1, 2, 3, 4], [1, 2, 3, 4]) == [0, 0, 0, 0]
68
69 # Match test case 4: Test identical arrays
70 # Returns an array with correct values when input arrays are of length 2
    Codiumate: Add more tests
71 def test_correct_values_length_2(self):
72     assert compare([1, 2], [1, 2]) == [0, 0]
73
74 # Match test case 7: Test with no match element
75 # Returns an array with correct values when input arrays are of length 3
    Codiumate: Add more tests
76 def test_compare_length_3(self):
77     assert compare([0, 5, 0], [4, 1, 1]) == [4, 4, 1]
78     assert compare([10, 20, 30], [5, 10, 15]) == [5, 10, 15]
79
80 # Match test case 6: Test with no match element
81 # Returns an array with absolute difference values when input arrays are different
    Codiumate: Add more tests
82 def test_arrays_with_different_values(self):
83     assert compare([1, 2, 3, 4, 5, 1], [1, 2, 3, 4, 2, -2]) == [0, 0, 0, 0, 3, 3]
84     assert compare([0, 5, 0, 0, 0, 4], [4, 1, 1, 0, 0, -2]) == [4, 4, 1, 0, 0, 6]
85
86 # Match test case 3: Test with one element
87 # Works correctly with input arrays of length 1
    Codiumate: Add more tests
88 def test_works_correctly_with_input_arrays_of_length_1(self):
89     assert compare([1], [1]) == [0]
90
91 # Works correctly with input arrays of length 1000
    Codiumate: Add more tests
92 def test_works_correctly_with_input_arrays_of_length_1000(self):
93     game = [1] * 1000
94     guess = [1] * 1000
95     assert compare(game, guess) == [0] * 1000
96
97 # Duplicate test case 6
98 # Works correctly with input arrays of small integer values
    Codiumate: Add more tests
99 def test_compare_small_integer_values(self):
100     assert compare([1, 2, 3, 4, 5, 1], [1, 2, 3, 4, 2, -2]) == [0, 0, 0, 0, 3, 3]
101     assert compare([0, 5, 0, 0, 0, 4], [4, 1, 1, 0, 0, -2]) == [4, 4, 1, 0, 0, 6]

```

Figure B.106: Sample 30: Relevant Test Cases, part 2/2

