COMP 354 Software Engineering

LCL123: **LEARNING CODING LIKE 1,2,3**

ASSIGNMENT 4: PLANNING AND EXECUTION OF THE SOFTWARE DEVELOPMENT PROJECT FOR ALGORITHM DEVELOPER SOFTWARE

FALL 2021

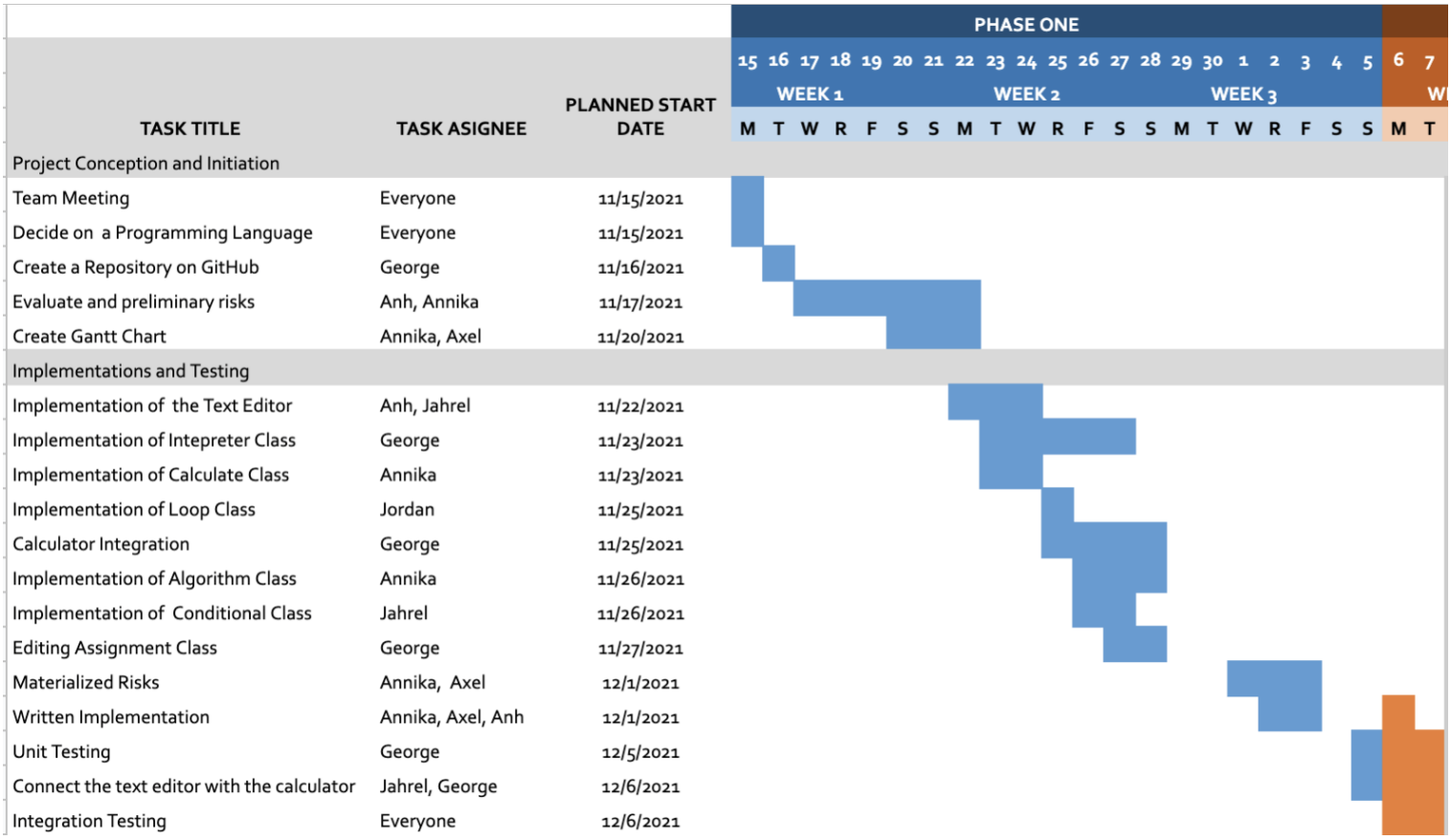
Annika Timermanis
George Mavroeidis
Jahrel Stewart
Axel Solano
Phuong Anh Trinh
Jordan Chan Kum Sang

December 2nd, 2021

Dr. Rajagopalan Jayakumar
Farbod Farhour

Schedule

Gantt Chart



TASK TITLE	TASK ASIGNEE	PLANNED START DATE
Project Conception and Initiation		
Team Meeting	Everyone	11/15/2021
Decide on a Programming Language	Everyone	11/15/2021
Create a Repository on GitHub	George	11/16/2021
Evaluate and preliminary risks	Anh, Annika	11/17/2021
Create Gantt Chart	Annika, Axel	11/20/2021
Implementations and Testing		
Implementation of the Text Editor	Anh, Jahrel	11/22/2021
Implementation of Intepreter Class	George	11/23/2021
Implementation of Calculate Class	Annika	11/23/2021
Implementation of Loop Class	Jordan	11/25/2021
Calculator Integration	George	11/25/2021
Implementation of Algorithm Class	Annika	11/26/2021
Implementation of Conditional Class	Jahrel	11/26/2021
Editing Assignment Class	George	11/27/2021
Materialized Risks	Annika, Axel	12/1/2021
Written Implementation	Annika, Axel, Anh	12/1/2021
Unit Testing	George	12/5/2021
Connect the text editor with the calculator	Jahrel, George	12/6/2021
Integration Testing	Everyone	12/6/2021

Risk Analysis

Descriptions, Criticalities and Mitigation Plans

We will outline the known and predictable risks associated with the implementation of LCL123 software.

Risk 1

Risk: The first predictable risk is related to *staff experience*. The developers' experience in GUI application in Python is limited so that could result in the development time of the Controller, View components taking longer than anticipated.

Criticality: This is a *marginal* risk.

Mitigation: The mitigation plan for Risk 1 is ensuring the deadline of completion for the editor.py (UI file) needs to be completed as early as possible to anticipate delays when learning and implementing the GUI.

Risk 2

Risk: The second predictable risk is about the *developer environment*. Some members are still unfamiliar with GitHub since they do not use them regularly.

Criticality: This is *marginal* risk.

Mitigation: The mitigation plan for Risk 2 is that team members who don't have as much experience working with GitHub will be assigned to watch tutorials on YouTube such as "Git and GitHub for Beginners - Crash Course".

Risk 3

Risk: The third predictable risk is about the *delivery deadline*. Even with strong team communication, unanticipated issues with the code implementation could cause a very tight deadline.

Criticality: This is a critical risk.

Mitigation: The mitigation plan for Risk 3 is to set earlier deadlines for each Class code to be completed, in order to prepare in cases of longer debugging times.

Risk 4

Risk: The fourth predictable risk is about the *end user*. While developing our software GUI, our GUI might function correctly, however it might not be as interpretable for a child as it is for one of our developers.

Criticality: This is a *marginal* risk.

Mitigation: The mitigation plan for Risk 4 is to have a child or younger sibling test run the program to incorporate the target user's perspective.

Risk 5

Risk: The fifth predictable risk is about the *connectedness of the front end UI and the backend*.

Criticality: This is a *catastrophic* risk.

Mitigation: The mitigation plan for Risk 5 is to ensure strong communication between team members working on front end UI and backend code.

Risk 6

Risk: The sixth predictable risk is about the *cohesion and coupling* of our code.

Criticality: This is a *critical* risk.

Mitigation: The mitigation plan for Risk 6 is to run test cases for each class as it is coded, and integrate our later test cases as we progress. It is possible that our architectural design, even after our revision in Assignment 3, could still have too many dependencies when actually being implemented that would cause errors and defects. We must mitigate this risk by actively testing

the code and ensuring strong communication between team members responsible for each class.

Materialized Risks

In this part, we will analyze the risks from above, and present if they materialized throughout the construction process, if our planned mitigation strategy worked, and the ultimate consequences of each risk.

Risk #	Did It Materialize	Did Our Mitigation Strategy Work	Consequences
1	Yes	Somewhat	The editor.py file was completed in the early stages of our progress as can be seen in the Gantt Chart listed above as was listed in our mitigation strategy for Risk 1, however the connectedness of front end editor and back end code was still an issue that arose, despite the earlier completion of the editor file.
2	Yes	Yes	As a team we all took on the responsibility to familizare with GitHub before starting the project code as listed in our original mitigation strategy for Risk 2. George is the one who set up the repository. To prevent conflicts when merging, we set up a development branch and a main branch, where only Jahrel has access to the main to merge what is needed to ensure everyone's individual contributions were merged smoothly.
3	Yes	Somewhat	Overall, setting earlier deadlines for class codes to be completed helped mitigate this risk, however the software completion still came close to the final deadline.
4	Yes	No	Our mitigation strategy was not conclusive since we did not have time to have a target user try the LCL123 software before the deadline was due.

5	Yes	Yes	Good communication helped mitigate the potential catastrophic behaviour of Risk 5. We still had issues arise when trying to connect the UI with the backend code, however they were more manageable, and the communication between team members helped.
6	Yes	Yes	The component level tests while coding prevented a lot of errors during the merging of the components. Therefore, the integration tests that were previously designed uncovered very few errors in the code.

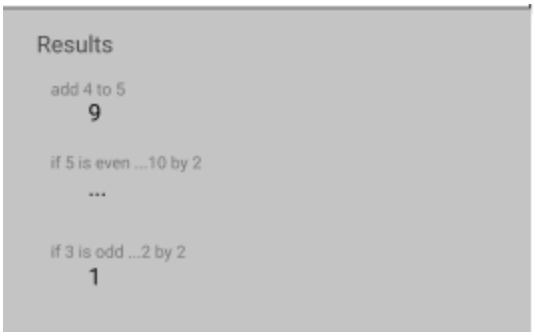
Implementation

Changes During Construction Phase and How These Changes Improved the Quality of Software

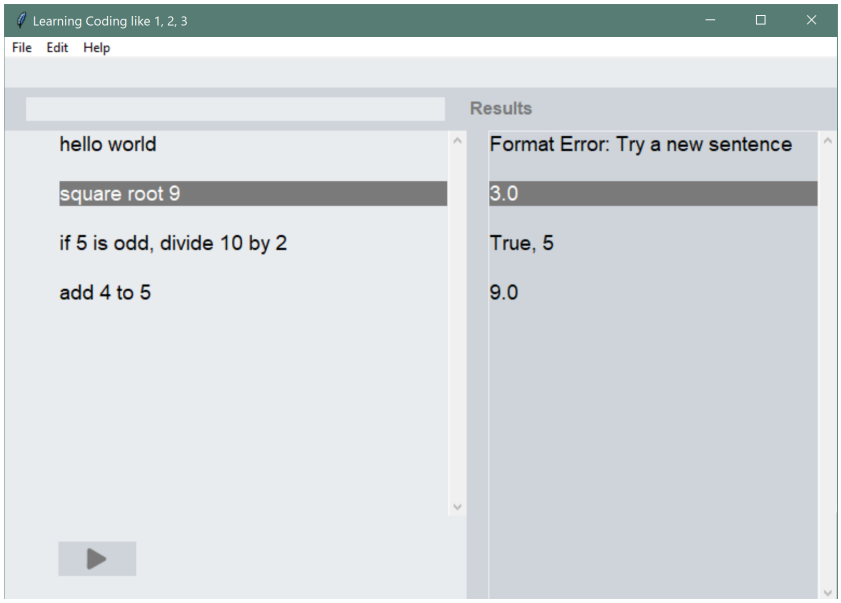
There will be some changes in our UI design.

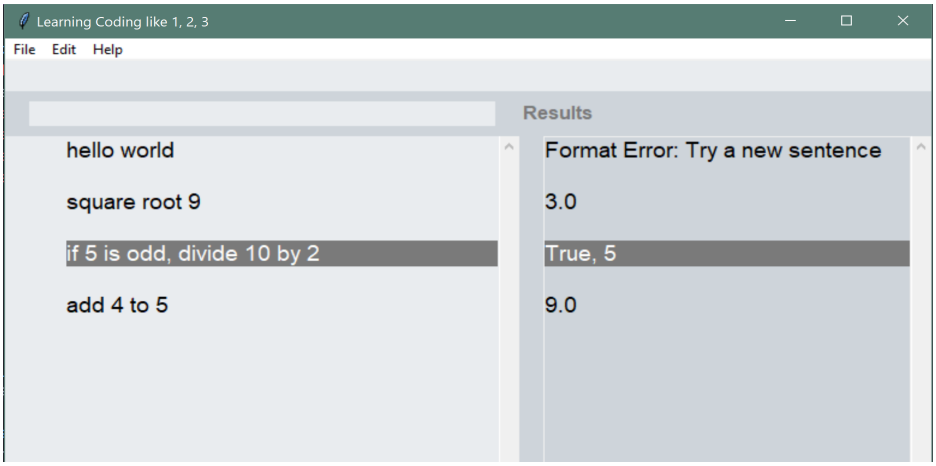
Change 1

In our initial design, in the results area, there are both the input sentence and its result to be shown.



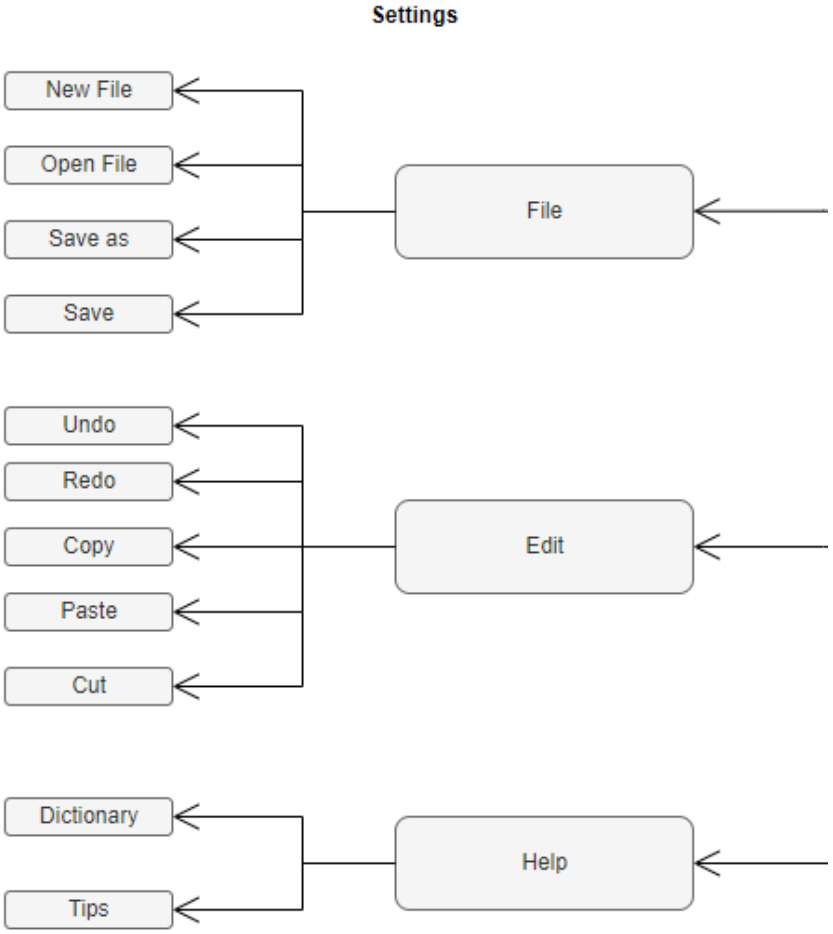
After discussing with the stakeholders and developers of the team, we realized the difficulty of doing so, therefore, instead of having both the input sentence and its result in the results area, we only have results in the results area. And when we want to see the result of any inputted sentences, we just need to click the sentence in the history and the corresponding result in the result area will be highlighted. For example:



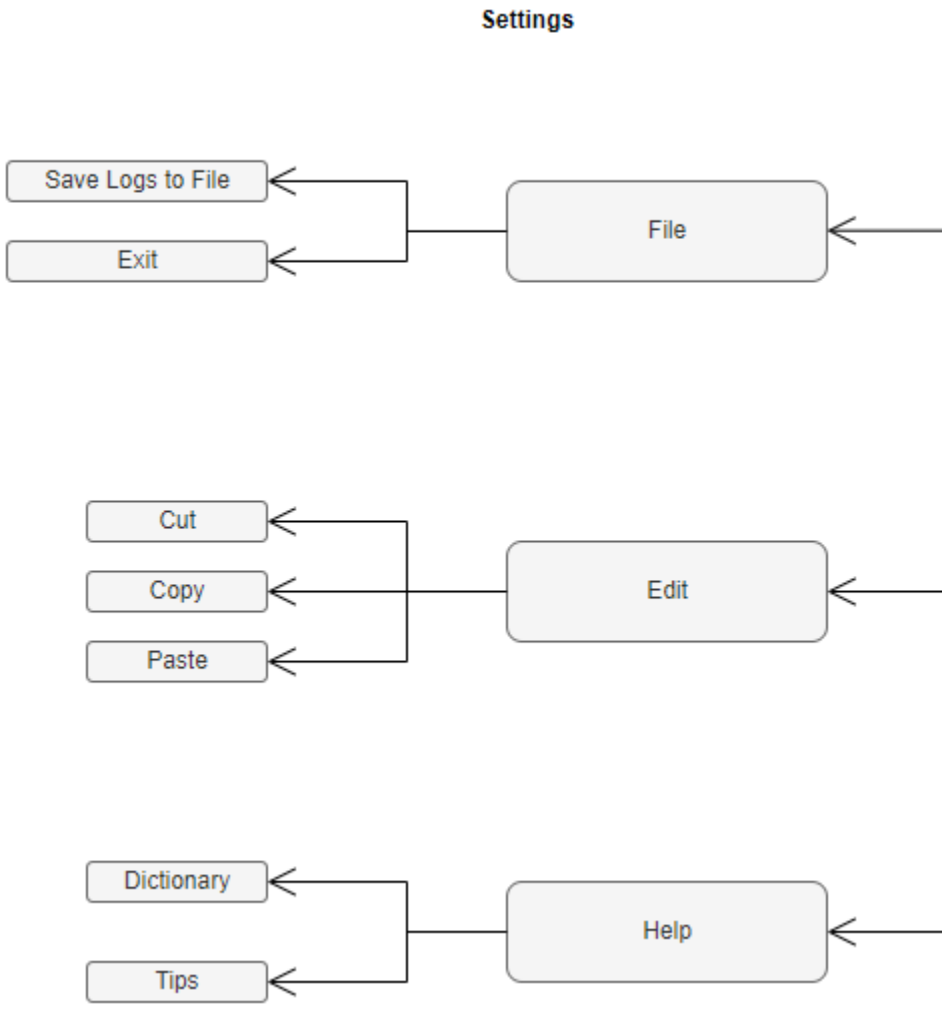


Change 2

In our initial design, the features of the text editor include as the following:



After discussing with the stakeholders and developers, we all agree to remove some unnecessary features, instead, only those are shown in the text editor. It will make the text editor minimal and very child-friendly.



How Each Component in our Design was Implemented & Tested

Each component of the calculator was implemented in an object-oriented manner, as described in Assignment 3. At component level, we tested each class with component level test cases which were previously written in Assignment 3. The structure of our class implementations closely followed our outline UML diagram from Assignment 3.

For the UI component, we used the library **Tkinter** in order to implement the text editor. We used `grid()` to put each component to the desired area. There are two main areas of the text editor: sentences area and results area. There are frames, labels, list boxes inside each area so that the layout can be similar to the design as much as possible.

How Each Component in our Design was Integrated Into the Complete Software

The order of implementation of each component can be precisely seen in our gantt chart above. We focused on writing and testing the Calculate, Loop, Conditional and Assignment classes, exclusively, running the component level test cases on each to ensure they were working. The interpreter class was then written to be a tool to help split up the input that would be fed through to the Algorithm Class. For example,

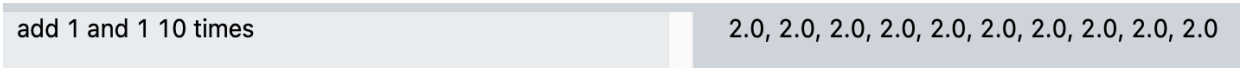
The Algorithm class required that the calculate, loop, conditional and assignment classes be implemented, in order to implement the function `findAlgorithm()`. The algorithm class also has a set of methods such as `isNumber()` and `isVariable()` in order to validate the input and convert the string values to numerical values.

After the calculator (backend) works well, we have to connect it with the text editor (front end). We will get the content of the sentenceInputBox and call the methods inside the

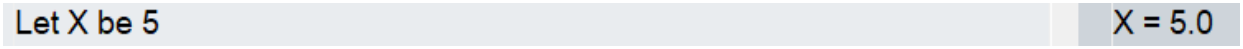
Interpreter class to split the clauses and find algorithms in our calculator, then the corresponding result that the calculator processed will be appended to the result area. All processes are through `process_Algorithm(self, sentence)` in `editor.py`. When the user inputs a sentence, they just need to to hit “Enter” or click “Run” button so that the result will show.

Test Cases

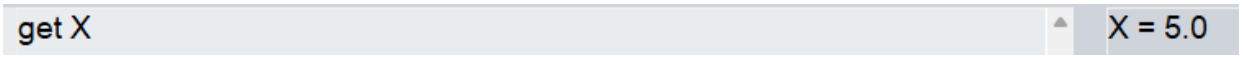
Input: add 1 and 1 10 times → 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0



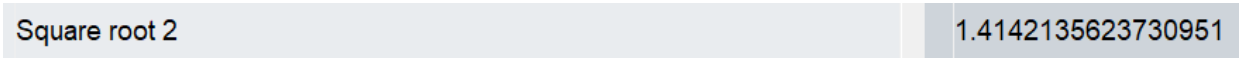
Input: Let X be 5 → X = 5



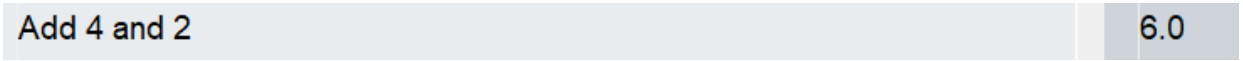
Input: get X → X = 5



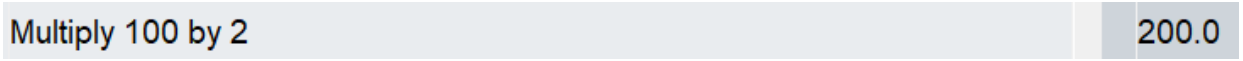
Input: Square root 2 → = 1.4142…



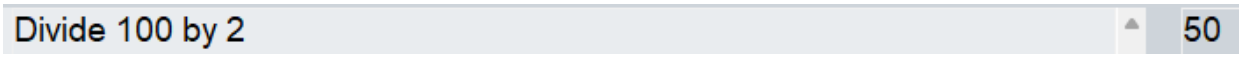
Input: Add 4 and 2 → = 6



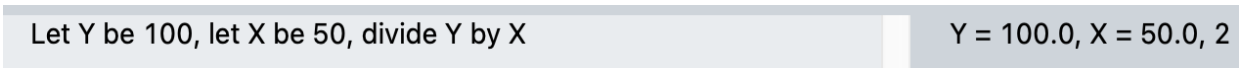
Input: Multiply 100 by 2 → = 200



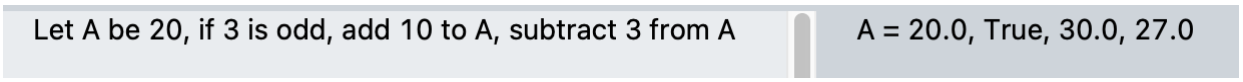
Input: Divide 100 by 2 → = 50



Input: Let Y be 100, let X be 50, divide Y by X → = 2



Input: Let A be 20, if 3 is odd, add 10 to A, subtract 3 from A → A = 27

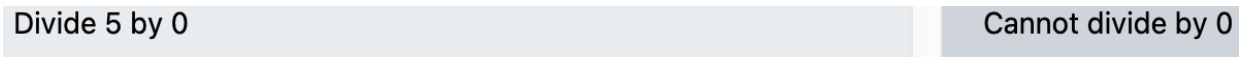


Input: Let X be 95, if 4 is odd, add 10 to X, subtract 5 from X, divide X by 2

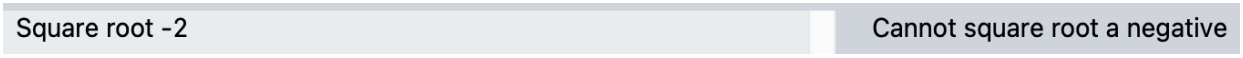
Input: Let X be 200, let Y be 50, add 5 to X 2 times, let Y be X, add 2 to Y

Error Cases

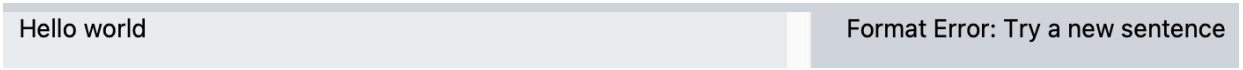
Input: Divide 5 by 0 → Cannot divide by 0



Input: Square root -2 → Cannot square root a negative



Input: Hello world → FormatError: Try a new sentence



Input: If 5 is odd → No statement found after condition

Input: “ if 6 is odd, divide 6 by 3 → False Condition. Statement skipped

if 6 is odd, divide 6 by 3	False condition. Statement skipped
if 6 is odd, divide 6 by 3	False condition. Statement skipped

Input: let y be x → (where variable does not exist)

let y be x	Variable does not exist

References

[1] R. Pressman and B. Maxim, Software Engineering: A Practitioner's Approach, 9th Edition, McGraw Hill, 2020, ISBN 13: 9781259872976.