Developing Testing-First Labs For a Less Intimidating Introductory CS Experience

Angela Zavaleta Bernuy
Dept. of Computer and Mathematical Sciences
University of Toronto Scarborough
angelazb@cs.toronto.edu

Brian Harrington
Dept. of Computer and Mathematical Sciences
University of Toronto Scarborough
brian.harrington@utoronto.ca

Abstract

When introducing non-majors to programming in an introductory computer science course, the simple mechanics of code writing can be intimidating. Many students report feeling overwhelmed by the requirements of user interfaces and syntax guides before even writing their first line of code. In an attempt to combat this anxiety, we have developed a tool called *Code Detective*, which allows students to learn fundamental skills of computer science: testing, program description, debugging and tracing before ever having to write any code.

Code Detective starts by completely hiding the code, asking students to reverse engineer the specifications of each module from only the inputs and outputs. Over several weekly laboratory sessions, students are then introduced to program definition, documentation and testing, as more elements of the code are revealed. Students then learn tracing and debugging, all before actually being required to directly write or edit any code.

In this experience report, we discuss the development and deployment of Code Detective in an Introduction to Programming course for non-majors course at a large North American research university.

1. Introduction

Learning to program is perceived by many students as a very challenging academic task (Bennedsen & Caspersen, 2007). It is a common understanding that students can get intimidated and overwhelmed when they are introduced to programming, especially if they have a fixed idea on their minds about the levels of difficulty of the subject.

Reports show that there are high failure rates when learning to program, and many students choose not to take computer science courses because they find the concept of programming to be intimidating (O'Donnell, Buckley, Mahdi, Nelson, & English, 2015). Educators have been trying to improve the overall students' satisfaction as it is connected with students' retention (Rybarczyk, 2020). As non-majors have different learning habits from traditional computer science students (Rybarczyk, 2020), increasing engagement during a non-majors class is always a challenge (O'Donnell et al., 2015).

Previous work about other strategies to introduce programming includes educational games, breadth-first approaches, testing first, among others. Educational games have been used to increase students' engagement and retention (Lee, Ko, & Kwan, 2013). Another approach is breadth-first, which includes teaching students everyday computer tasks like image editing, OS installation and building home computer networks in an undergraduate course (McFall & DeJongh, 2011). There has been work done in testing-first approaches to introductory programming. Marrero Settle conducted a course where students were required to implement their test cases before completing their assignments (Marrero & Settle, 2005).

This work explores an alternative way of introducing programming to non-major students that combines some elements of breadth-first and testing-first approaches, utilizing more traditional deductive reasoning skills in place of technical abilities that may be new and intimidating to novice programmers. The focus of this work is to present programming as a deductive logical process first, allowing students to think algorithmically, and become comfortable with fundamental concepts of programs and functions, before presenting them with actual code. In this way, students can begin by using tools and methods with

which they are already familiar while they are gradually introduced to more specific computer science concepts.

2. Code Detective

Code Detective is a tool that was designed as part of an Introduction to Programming course for non-major students at The University of Toronto Scarborough. The web tool was developed by a team of undergraduate computer science students. Code detective focuses on introducing non-major students to the concept of computing in a way that emphasizes skills they already possess, without forcing them to write (or initially, even read) code that they may find overwhelming. The programming languages taught in the course were Scratch and Python, and this tool was easily adaptable to both languages.

Code Detective consists of nine modules that are aligned with the material covered during the weekly lectures starting on the second week of the semester. Students worked in pairs on the Code Detective modules during two-hour weekly laboratory sessions supervised by teaching assistants. The teaching assistants were mainly tasked with providing guidance to groups as needed. At the end of each session, students were asked to present their solutions and explain their reasoning.

Each module consists of a series of questions about various programs with simple logic. In the first modules, the program's code is entirely hidden, only offering input and output on the screen where the students are asked to experiment with the program and deduce what it does, formally define the program's function, and develop a testing plan to determine if the program has any bugs. For later modules, students practice how to trace and repair code, without the need for writing any code. Only in the later modules are students asked to write or edit code.

The nine modules were created following a gentle, yet increasingly difficult pedagogy as follows:

2.1. Lab 0: Program definitions

Consists of twenty questions. Each question has a small program with a series of input/output boxes (check-boxes, text boxes, date selectors). The students need to provide some input, click the "run" button and keep track of the response. After experimenting with the program, they are asked to provide a formal definition for the function which includes: stating what the function does, valid input and expected output.

2.2. Lab 1: Program definition and introduction to algorithms

Consists of two parts, five questions each. For the first five questions, the students are given the definition of a program and a high level algorithm that would implement the definition in a flawed way. Their goal is to find the flaws in the algorithm by providing a list of test cases that will fail and write the fixed algorithm. For the last five questions, the students only get the definition of a program and they have to provide an algorithm to solve the problem.

2.3. Lab 2: Test dimensions and black-box testing

Consists of two parts, five questions each. For the first five questions, the students were provided with the definition of a program and are tasked with designing a testing plan. They need to provide the dimensions of the testing space, decide which are the important points on each dimension, and calculate the number of tests required for a full coverage testing. For the last five questions, they need to perform black-box testing of a given program and provide a list of failed test cases specifying the input, expected output, and actual output.

2.4. Lab 3: Tracing and white-box testing

Consists of two parts, five questions each. For the first five questions, the students are provided with a simple algorithm. Their task is to trace the code for a given input and enter the output. The students get immediate feedback from Code Detective and a live-count of failed attempts while entering their answers. For the last five questions, they need to perform white-box testing of a given Scratch program and provide a list of failed test cases specifying the input, expected output, and actual output.

2.5. Lab 4: Tracing and debugging

Consists of two parts, five questions each. For the first five questions, the students are provided with more complex code than the previous module as loops are introduced. Their task is to trace the code for a given input and enter the output. The students get immediate feedback from Code Detective and a live-count of failed attempts while entering their answers. For the last five questions, the students are given a broken program and their task is to debug and fix the code.

2.6. Lab 5: Refactoring and implementation

Consists of two parts, five questions in total. For the first three questions, the students get a working program that is not well designed. Their task is to refactor it by creating smaller, better designed modules without breaking the code. For the last two questions, the students get an incomplete program with missing code segments (missing blocks in Scratch). Their task is to implement the missing components to get the code working.

2.7. Lab 6: Efficiency, and implementation

Consists of two parts, five questions in total. For the first three questions, the students get a working program implemented inefficiently. Their task is to refactor the program without changing its functionality. For the last two questions, the students get the definition of a program with a set of specifications that they are required to implement.

2.8. Lab 7: Scratch-Python translation

Consists of five questions. Each question has five different Scratch working programs. The students need to translate the Scratch code into Python code. The students are required to test their translated code using an IDE and demonstrate their testing plan. This module is specifically designed for courses that cover more than one language in order to help students learn to work with a new language, but can also be used to help students understand how much of their learning is transferable to other languages.

2.9. Lab 8: Design and implementation

The students are provided with a partially completed Python code. They are required to read and understand the code, as well as to implement the missing documented functions.

Following the completion of the Code Detective modules, the students are required to work on a project of their choice where they had to implement a program either using Scratch or Python. Some of the most common projects were arcade games in Scratch and simple data management programs in Python.

3. Evaluations

Based on the anonymous course evaluations, some students shared that they enjoyed this course design because they did not have any prior coding experience and they were gradually exposed to the course content. Moreover, one student stated that *"this was good because it prevented me from getting scared off from programming forever"*. A couple of students shared that the structure of the course helped reduce intimidation as it helped them *"let go of the mindset that computer science is intimidating and instead makes us see that computer science can be for everybody"*, and that we created *"a learning atmosphere that was not intimidating as a person with no experience at coding!"*.

Many students stated that the course was still challenging and required time and effort to develop a deeper understanding of programming concepts. They agreed that Code Detective helped reinforce the lecture material and encouraged them to get more practice. On the other hand, students who had prior coding experience reported that they felt the class progress slow and they wished they were exposed to harder concepts.

To measure the success of Code Detective compared to previous deliveries of the same course, we looked at the drop rates from previous years. We found that the year in question had a 7% drop rate, around 23% lower than the previous years: 29% and 30% in the two years prior, even though the class size of 450 students was the same across the three years. While we cannot attribute this substantial reduction in drop rates to Code Detective directly, as other factors such as teaching staff and structure were not held

constant, the teaching team commented on the absence of the phenomenon, observed in previous course offerings, of students dropping after being unable to complete the first lab sessions.

4. Conclusion

Code detective played a key role in reducing students' sense of intimidation, and fostered a sense of accomplishment without resorting to paternalistic methodologies or games that may alienate some students. Introducing computer science as a deductive, logical, problem solving system first before introducing the technicalities of code writing made students feel that they were able to use the logic and reasoning skills they already possessed to solve problems, and gave them a gentler and less intimidating introduction to programming.

5. References

Bennedsen, J., & Caspersen, M. E. (2007, June). Failure rates in introductory programming. *SIGCSE Bull.*, *39*(2), 32–36. doi: 10.1145/1272848.1272879

Lee, M. J., Ko, A. J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. In *Proceedings of the ninth annual international acm conference on international computing education research* (pp. 153–160).

Marrero, W., & Settle, A. (2005). Testing first: emphasizing testing in early programming courses. In *Proceedings of the 10th annual sigcse conference on innovation and technology in computer science education* (pp. 4–8).

McFall, R. L., & DeJongh, M. (2011). Increasing engagement and enrollment in breadth-first introductory courses using authentic computing tasks. In *Proceedings of the 42nd acm technical symposium on computer science education* (pp. 429–434).

O'Donnell, C., Buckley, J., Mahdi, A., Nelson, J., & English, M. (2015). Evaluating pair-programming for non-computer science major students. New York, NY, USA: Association for Computing Machinery.

Rybarczyk, R. (2020). Non-major peer mentoring for cs1. In *Proceedings of the 51st acm technical symposium on computer science education* (p. 1068–1074). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3328778.3366901

6. Appendix: Code Detective Interface

Figure 1 – ***Code Detective Main Menu***

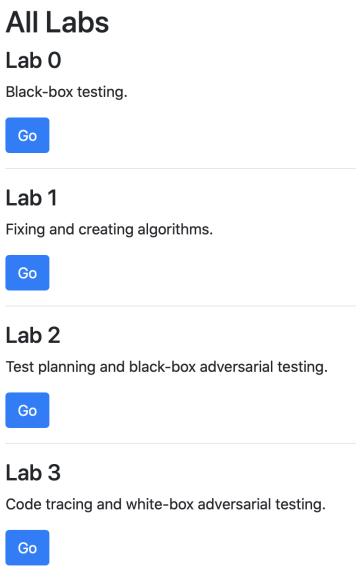


*Figure 2 – **Lab 0 Menu***

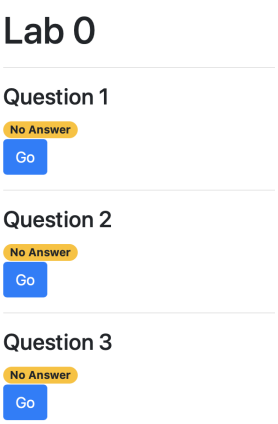


*Figure 3 – **Lab 0 Sample Question***

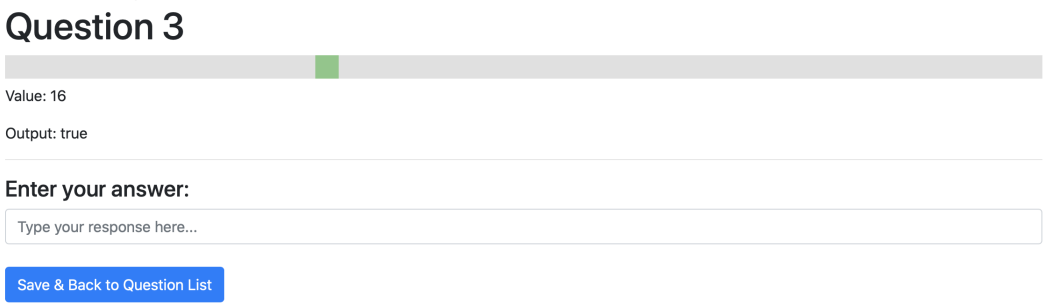


*Figure 4 – **Lab 1 Sample Question 1***

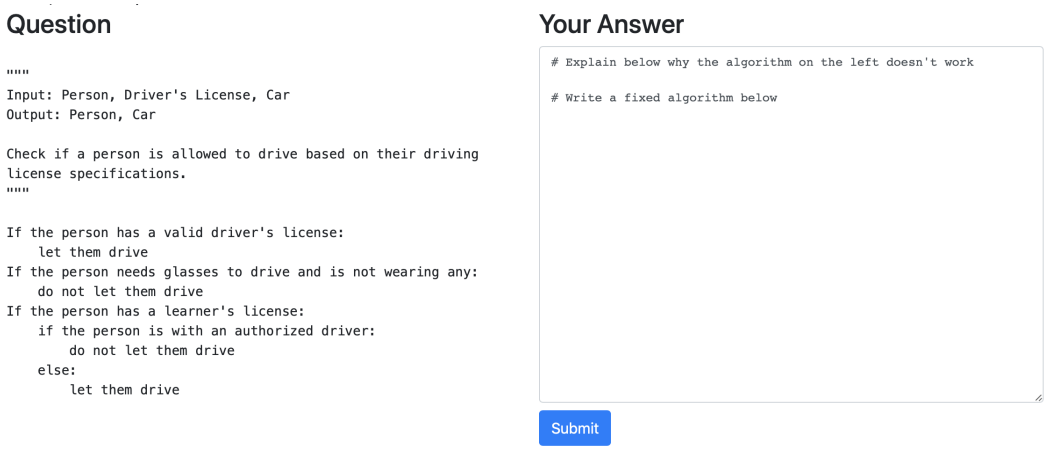


*Figure 5 – **Lab 1 Sample Question 2***

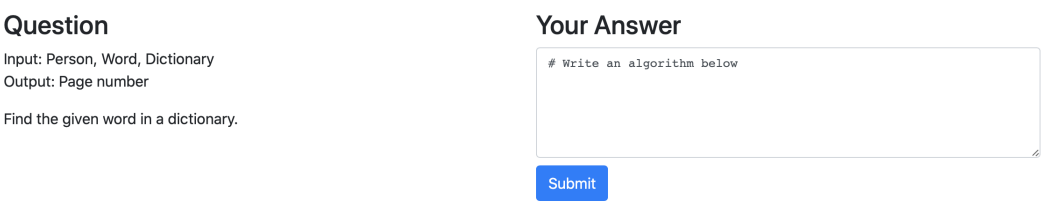


*Figure 6 – **Lab 2 Sample Question 1***

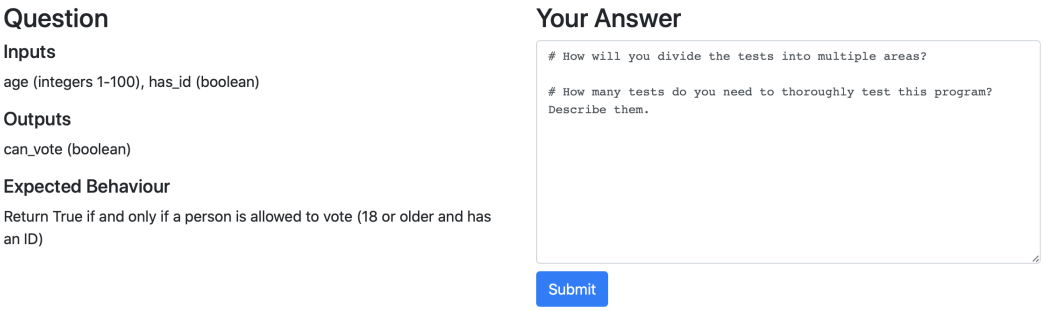


*Figure 7 – **Lab 2 Sample Question 2***

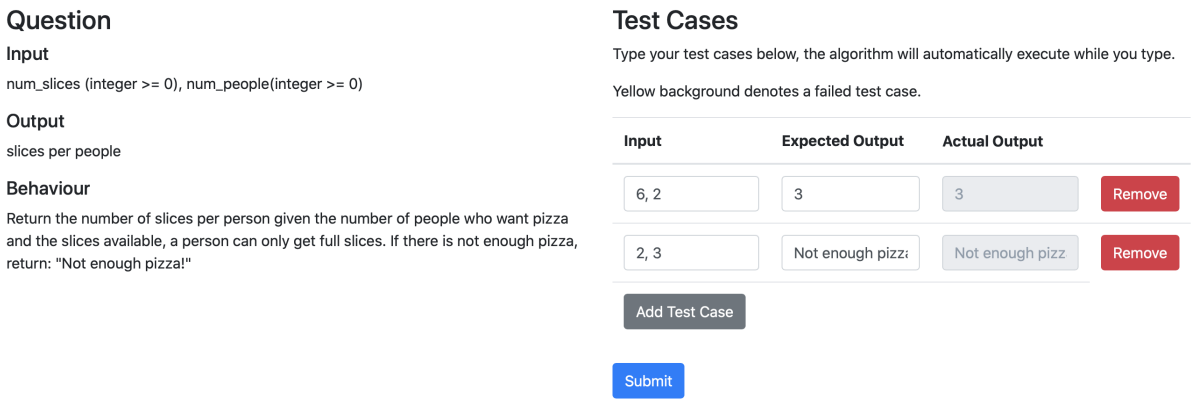


*Figure 8 – **Lab 3 Sample Question 1A***

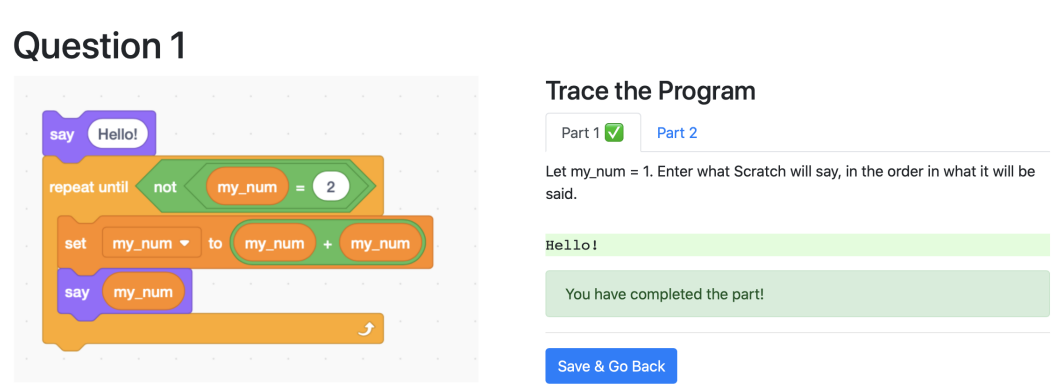


*Figure 9 – **Lab 3 Sample Question 1B***

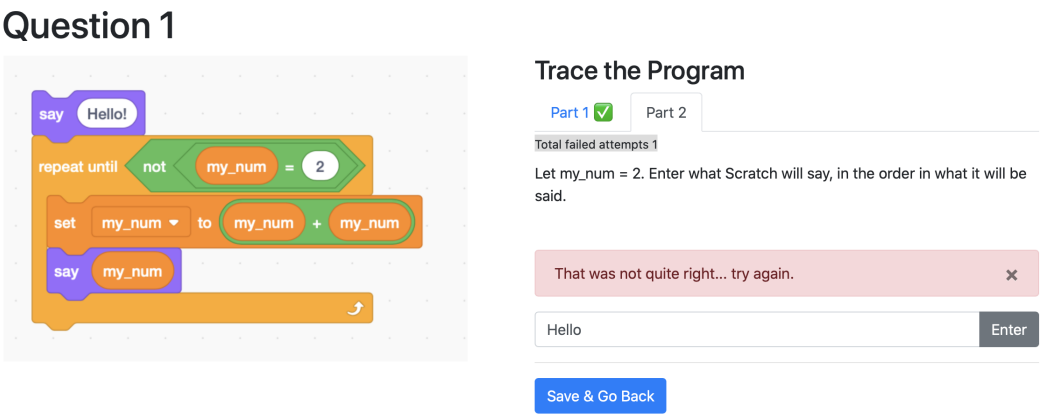


*Figure 10 – **Lab 3 Sample Question 2***

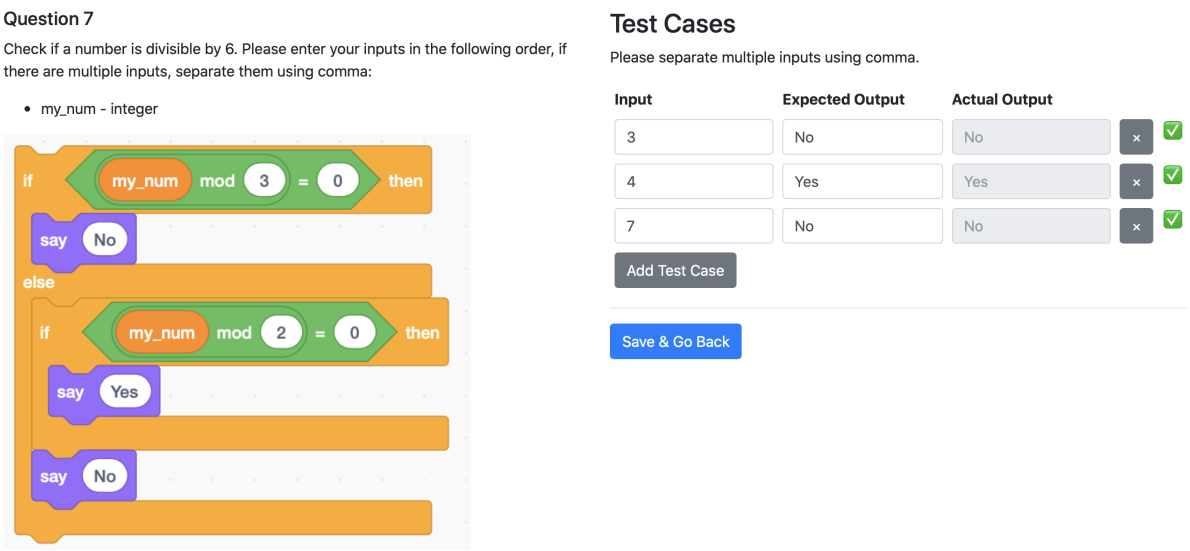


*Figure 11 – **Lab 4 Sample Question***

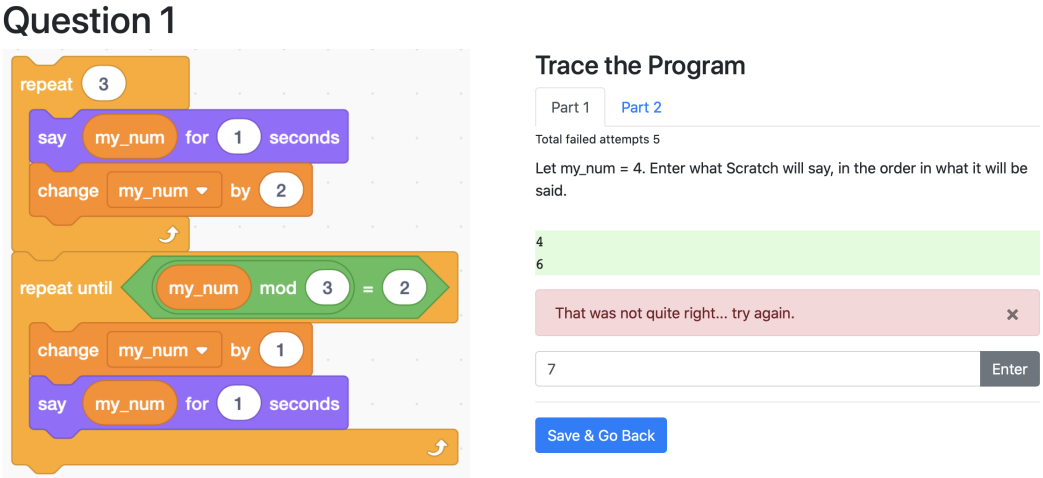


*Figure 12 – **Lab 5 Sample Question***

Question 2

Carefully read the description below, and fix the program on Scratch. When you open the project, click on "Remix" to copy the project into your account and fix the changes there.

Implement the block perimeter. Hint: Try using the instruction "item # of ___ in shapes"

- Input: shape (triangle, square, pentagon, hexagon, heptagon, or octagon)
- Output: perimeter

Go to Scratch

*Figure 13 – **Lab 6 Sample Question***

Question 2

Carefully read the description below and complete the question on Scratch. When you open the project, click on "Remix" to copy the project into your account and fix the changes there.

Task: Improve the code without changing the actual functionality.

- Input: word (string)
- Output: has_digits, has_symbols, has_vowels

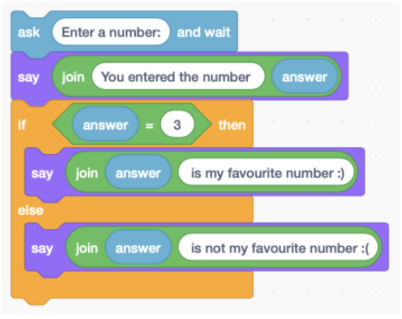
Check if the input has at least one digit, one symbol or one vowel. If it has any of these, change the appropriate variable to "Yes", if not, leave it as "No"

Go to Scratch

*Figure 14 – **Lab 7 Sample Question***

Question 1

Read through the Scratch code below and re-write it in Python. You may use any editor you want, after you have completed the task, paste your Python code below.



Enter your Python code here...

Submit & Go Back

Figure 15 – ***Lab 8 Sample Question***