The code is lava: Improving children's debugging skills with explicit instruction

by

Christopher Patrick Kerslake

B.Sc. (Computing Science), Simon Fraser University, 1996

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Arts

in the
Educational Technology and Learning Design Program
Faculty of Education

© Chris Kerslake 2023
SIMON FRASER UNIVERSITY
Summer 2023

Copyright in this work is held by the author. Please ensure that any reproduction
or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Christopher Patrick Kerslake**
Degree:	**Master of Arts (Education)**
Title:	**The code is lava: Improving children's debugging skills with explicit instruction**
Committee:	**Chair:** Cary Campbell Limited Term Lecturer, Education
	D. Kevin O'Neill Supervisor Professor, Education
	Kristiina Kumpulainen Committee Member Professor, Education
	Parmit Chilana Examiner Associate Professor, Computing Science

Ethics Statement

The author, whose name appears on the title page of this work, has obtained, for the research described in this work, either:

a. human research ethics approval from the Simon Fraser University Office of Research Ethics

or

b. advance approval of the animal care protocol from the University Animal Care Committee of Simon Fraser University

or has conducted the research

c. as a co-investigator, collaborator, or research assistant in a research project approved in advance.

A copy of the approval letter has been filed with the Theses Office of the University Library at the time of submission of this thesis or project.

The original application for approval and letter of approval are filed with the relevant offices. Inquiries may be directed to those authorities.

Simon Fraser University Library
Burnaby, British Columbia, Canada

Update Spring 2016

Abstract

The debugging strategies of nine elementary students were investigated during an eight-week Python programming course which integrated explicit debugging instruction. A verbal think-aloud protocol was used to document the strategies students used to solve debugging challenges. Analysis of the data involved content coding that combined codes derived from Vessey (1984) with inductively-developed codes necessary to cover the data. Three groups of students were identified. The first group adopted the explicit debugging strategies taught in the course and used them successfully but exhibited signs of excessive cognitive load. The second group used the strategies sparingly, and instead relied on their own expert-like strategies. The third group did not use the taught strategies, did not understand their role, and ultimately gave up during the final challenge. The findings of the study suggest that students who adopted the explicit debugging strategies were successful but appear to require additional practice to master them.

Keywords: debugging; computer science education; novices; computer programming; python

Dedication

This thesis is dedicated to all of the teachers who have patiently listened to an eager student talk about their ideas, encouraged them to follow them, and supported them along the way. Thank-you.

Acknowledgements

I would like to thank my wife Lisa for her unconditional love and support throughout this project. This thesis would not have been possible without your encouragement to pursue this idea, and your unwavering love and support throughout.

I would like to express my sincere gratitude to my supervisor Dr. D. Kevin O'Neill. What started out as an afternoon coffee in 2018 discussing my interest in the teaching of computer programming has resulted in a new life direction because of your patience and guidance throughout.

I would also like to thank my committee member Dr. Kristiina Kumpulainen. Thank-you for your insights on the research process, your focus on the research questions, and your encouragement throughout.

I want to thank the students (and their parents) who participated in this study. Thank-you for committing the time necessary to participant in the study, without your support and participation, this thesis would not have happened.

I am grateful to the three people who acted as my references for admission to the SFU ETLD master's program: Bill, Noreen, & Ebru. Bill who encouraged me to seek out the program. Noreen & Ebru who supported me in my early endeavours at Bayview Community School and then supported my application to the program. Without your support, I would not have been able to start down this path.

Finally, I would like to thank my parents who supported me through it all. Thank-you for encouraging and supporting me throughout the years.

Table of Contents

Declaration of Committee ii
Ethics Statement iii
Abstract iv
Dedication v
Acknowledgements vi
Table of Contents vii
List of Tables x
List of Figures xi
List of Acronyms xiii
Glossary xiv

Chapter 1. Introduction 1
1.1. Study Background 1
1.2. Research Problem 5
1.3. Research Questions 5
1.4. Research Significance 6
1.5. Research Limitations 6
1.6. Organization of the Thesis 8

Chapter 2. Literature Review 9
2.1. Introduction 9
2.2. Definitions 10
2.2.1. Debugging Terms 10
2.2.2. Debugging Processes 13
2.2.3. Novices versus Experts 15
2.2.4. Debugging Code Written by Others 16
2.2.5. Turtle Graphics 17
2.2.6. Verbal Protocol Analysis 19
2.3. History of Debugging Research 20
2.3.1. A Call to Teach Debugging 20
2.3.2. Initial Debugging Research 21
2.3.3. Criticism of Early Research Methods 22
2.3.4. Verbal Reporting as Data 24
2.3.5. Explicit Debugging Instruction with Children 28
2.3.6. A Renewed Interest in Debugging 31
2.3.7. Debugging Instruction in the Classroom 35
2.3.8. Debugging-First: A New Approach to Teaching Debugging 40
2.4. Literature Review Limitations 41
2.5. Research Topic Justification 42

Chapter 3. Python Turtle Graphics Course Design 45
3.1. Classroom Programming Environment 47

3.2. Week 1: Study Introductions and Initial Debugging Challenges 49
3.3. Week 2: Pens, Colors, and Student Generated Examples 52
3.4. Week 3: For-Loops and Variables 54
3.5. Week 4: Shapes: Stars, Crosses, and Challenges 57
3.6. Week 5: Turtle Stars, Functions, and Challenges 58
3.7. Week 6: Bug Localization with the Wolf Fence Algorithm 59
3.8. Week 7: Complex Shapes with Functions 60
3.9. Week 8: Complex Shape: Among Us Character (Video) 62

Chapter 4. Methodology 64
4.1. Research Questions 64
4.2. Research Framework 64
4.3. Research Setting 65
4.4. Recruitment 66
4.5. Debugging Environment 66
4.6. Debugging Challenge Bugs 67
4.6.1. Week 1: Debugging Challenge #1 68
4.6.2. Week 1: Debugging Challenge #2 68
4.6.3. Week 1: Debugging Challenge #3 69
4.6.4. Week 1: Debugging Challenge #4 69
4.6.5. Week 2: Student-Generated Examples (SGE) 70
4.6.6. Weeks 4/5: Debugging Challenge 71
4.6.7. Weeks 7/8: Debugging Challenge 71
4.7. Student Questions 72
4.7.1. Student Self-Reported Self-Efficacy Scores 73
4.7.2. Student Debugging Description Question 73
4.7.3. Student Python Coding Description Question 73
4.7.4. Student Description of Debugging Difference with Expert 73
4.8. Data Collection 74
4.9. Data Analysis 76

Chapter 5. Findings 79
5.1. Research Questions 79
5.2. Data Sources 79
5.3. Participants 80
5.4. Week 1: One-on-one Debugging Sessions 81
5.4.1. Week 1 Student Debugging Strategies 81
5.4.2. Week 1 Student Debugging Challenges 85
5.4.3. Week 1 Debugging Times 87
5.4.4. Week 1 Pre/Post Self-Efficacy Ratings 89
5.5. Week 2: Student-Generated Examples and Expert Debugging 91
5.6. Weeks 4/5: One-on-one Debugging Challenges 92
5.6.1. Weeks 4/5 Student Debugging Strategies 92
5.6.2. Weeks 4/5 Student Debugging Challenges 96
5.6.3. Weeks 4/5 Debugging Times 98

5.6.4. Weeks 4/5 Pre/Post Self-Efficacy Ratings 100
5.7. Weeks 7/8: One-on-one Debugging Challenge 101
5.7.1. Weeks 7/8 Student Debugging Strategies 102
5.7.2. Weeks 7/8 Student Debugging Challenges 106
5.7.3. Weeks 7/8 Debugging Times 110
5.7.4. Weeks 7/8 Post Self-Efficacy Ratings 111
5.8. Summary of Student Self-Reported Self-Efficacy Ratings 112
5.9. Student Descriptions of Debugging 114

Chapter 6. Discussion 117
6.1. Research Aims & Research Questions 117
6.2. Summary of Key Findings 117
6.3. Discussion of Findings 118
6.3.1. Student Debugging Strategies 119
6.3.2. Student Debugging Challenges 122
6.3.3. Effects of Explicit Debugging Instruction 124
6.3.4. Student Responses To Explicit Debugging 128
6.4. Practical implications 132
6.5. Limitations of the study 134
6.6. Suggestions for Future Research 136
6.7. Concluding Summary 137

References 138

Appendix A. Code Book 145

Appendix B. Weekly Challenge Python Code 151
Week 1 Debugging Session Python Code (Challenges 1-4) 151
Week 1 – Challenge #1 – 2x Syntax Errors 151
Week 1 – Challenge #2 – 1x Syntax Error 151
Week 1 – Challenge #3 – 1x Indentation Error 152
Week 1 – Challenge #4 – 1x Semantic Error 153
Week 4/5 Debugging Sessions Python Code (4x semantic errors) 154
Week 7/8 Debugging Sessions Python Code (2x logic errors) 156

Appendix C. Student Debug Output Progression (Week 7/8) 159

Appendix D. Debugging Session Episodes 162

Appendix E. Recruitment Web Page 182

Appendix F. Parent Consent form 184

Appendix G. Student Assent form 187

List of Tables

Table 2.1. Nanja & Cook's expert, intermediate, and novice attribute rubric............28
Table 3.1. Study weekly outline 47
Table 5.1. Week 1 student debugging strategies 81
Table 5.2. Week 1 student debugging challenges..85
Table 5.3. Week 1 student self-reported self-efficacy ratings 90
Table 5.4. Week 2 students' observations of differences between the teacher (expert) and their debugging techniques (novices) 92
Table 5.5. Weeks 4/5 student debugging strategies..93
Table 5.6. Weeks 4/5 student debugging challenges 97
Table 5.7. Weeks 4/5 student self-reported self-efficacy ratings 101
Table 5.8. Weeks 7/8 student debugging strategies..102
Table 5.9. Weeks 7/8 student debugging challenges 107
Table 5.10. Weeks 7/8 student self-reported self-efficacy ratings 112
Table 5.11. Student descriptions of debugging (weeks 1, 2 & 6).............................116

List of Figures

Figure 2.1. PyCharm Community Edition integrated development environment. 11
Figure 2.2. Sample syntax error (missing parentheses) – Python 3.........................12
Figure 2.3. Sample runtime error (division by zero) – Python 3. 12
Figure 2.4. Sample semantic error – Python 3 Turtle Graphics. 13
Figure 2.5. Sample images generated during the course using turtle graphics.........18
Figure 2.6. Debugging flowchart from Klahr & Carver (1988, p. 378). 30
Figure 2.7. Debugging Tracing Heuristics (Fitzgerald et al., 2008, p. 114). 33
Figure 2.8. Michaeli & Romeike's three-level systematic debugging process. 38
Figure 2.9. Debugging strategy text (Ko et al., 2019, p. 471)..................................39
Figure 3.1. Sample images generated during the course using turtle graphics.........46
Figure 3.2. PyCharm Community Edition integrated development environment. 48
Figure 3.3. Sample turtle graphics output of a triangle. 49
Figure 3.4. Sample Python hello world program code. 50
Figure 3.5. Sample of bugged NameError version of print statement. 51
Figure 3.6. Sample NameError compiler error message (Week 1). 51
Figure 3.7. Week 2 sample brown coloured shape with green turtle........................53
Figure 3.8. Week 3 loops.py – print the loop values. 55
Figure 3.9. Week 3 for-loop progression (Python). 55
Figure 3.10. Week 3 Final octagon Python code with variables and a for-loop. 56
Figure 3.11. Week 3 debugging challenge. 57
Figure 3.12. Week 4 five-point outlined star shape using for-loops............................58
Figure 3.13. Multiple five-point connected-point stars created using a stars function. 59
Figure 3.14. Week 6 "four squares" debugging demonstration. 60
Figure 3.15. Week 7 turtle graphics flower example. 61
Figure 3.16. Weeks 7 & 8 final debugging challenge, the “Logo Bug Challenge”. 62
Figure 3.17. Week 8 three Among Us character drawn using turtle graphics. 63
Figure 4.1. Email recruitment statement. 66
Figure 4.2. Week 1 debugging challenge #1 buggy code. 68
Figure 4.3. Week 1 debugging challenge #1 corrected code. 68
Figure 4.4. Week 1 debugging challenge #2 buggy code. 69
Figure 4.5. Week 1 debugging challenge #2 corrected code. 69
Figure 4.6. Week 1 debugging challenge #3 buggy code. 69
Figure 4.7. Week 1 debugging challenge #3 corrected code. 69
Figure 4.8. Week 1 debugging challenge #4 buggy code. 70
Figure 4.9. Week 1 debugging challenge #4 corrected code (option 1). 70
Figure 4.10. Week 1 debugging challenge #3 corrected code (option 2). 70
Figure 4.11. Weeks 4/5 debugging challenge bugs side-by-side. 71

Figure 4.12. Weeks 7/8 commented out debug print-statements. 72
Figure 4.13. Partial debugging session transcript for participant Carl. 75
Figure 4.14. Sample debugging episode outline – Weeks 4/5 – George. 77
Figure 5.1. Screenshot of editor error highlights for challenge #1 (Week 1). 82
Figure 5.2. Screenshot of editor warning highlight for challenge #4 (Week 1). 82
Figure 5.3. Screen of sample compiler error for challenge #1 (Week 1). 83
Figure 5.4. Screenshot of Carl's edited code for challenge #4 (Week 1). 87
Figure 5.5. Screenshot of Carl's solution to challenge #4 (Week 1). 87
Figure 5.6. Week 1 debugging challenge times (in minutes, sorted by time). 88
Figure 5.7. Weeks 4/5 debugging challenge times (in minutes, sorted by time). 98
Figure 5.8. Peter's Week 4/5 debugging challenge episode trace. 99
Figure 5.9. Peter's Week 8 debugging challenge turtle output progression. 103
Figure 5.10. Brian's Week 8 debugging challenge turtle output progression. 104
Figure 5.11. George's Week 8 debugging challenge turtle output progression. 105
Figure 5.12. George's Week 8 debug output, printing values for *is_even* function. 105
Figure 5.13. Marks' Week 8 debugging challenge turtle output progression. 106
Figure 5.14. Oscar's Week 7 debugging challenge turtle output progression. 109
Figure 5.15. Lucas' Week 7 debugging challenge turtle output progression. 110
Figure 5.16. Weeks 7/8 debugging challenge times (in minutes, sorted by time). 111
Figure 5.17. Student self-reported debugging confidence scores. 113

List of Acronyms

API	Application programming interface. In this study, the turtle graphics API provided objects and functions that the programmers used to generate drawings on their computer screens.
CS	Computer science or computing science.
CS1	The first of the first two computer science (CS) courses typically taught to computer science majors, with CS2 being the second.
CS2	The second of the first two computer science (CS) courses typically taught to computer science majors, with CS1 being the first.
ETLD	Educational Technology and Learning Design. A graduate program at SFU focused on research-based educational technology and learning design.
IDE	Integrated development environment. A software program used for the creation and editing of computer programs.
ITiCSE	ACM conference on Innovation and Technology in Computer Science Education.
K-12	Kindergarten to grade 12. An expression used to refer to the publicly supported school grades prior to college in Canada and the United States.
PDF	Portable document format. A file format created by Adobe that preserves the layout of a digital document on any compatible reader.
SFU	Simon Fraser University.
URL	Uniform resource locator. A URL is a computer-recognizable address that can be used by a computer program like a web browser to navigate to the resource. For example, to visit the SFU library catalog, the URL is https://www.lib.sfu.ca/.

Glossary

Debugging	The process of identifying, locating, and repairing defects within a computer program.
Episode	“A group of task assertions related to the same goal or objective.” (Vessey, 1984, p. 268)
Expert	An individual who has extensive experience with a subject of study or field of work.
Guess-and-check	A debugging technique where the participant guesses at the cause of the problem, makes a change based on this guess, and then runs the program to check if their guess was correct or not.
Library (as in Software Library)	A software library is pre-written code that provides specific functionality that can be imported into a programming project and then used to provide the project with the library’s functionality. For example, a graphics library is imported so the program can output graphics.
Novice	An individual who is new to a subject of study or field of work.
PyCharm	The integrated development environment used by the participants in this study, created by JetBrains.
Python	A computer programming language used in this study, created by Guido van Rossum, and first released in 1991.
Zoom	An online video-conferencing system used in this study that provides for the sharing of audio and video for teaching as well as the sharing of computer screens.

Chapter 1.

Introduction

Debugging computer programs accounts for up to 80% of the time a programmer spends writing them (Ko & Myers, 2005; Mathis, 1974; O'Dell, 2017). As a result, a number of studies have been conducted over the past 50 years looking at how to reduce the time spent on debugging (Gould & Drongowski, 1974; Litecky & Davis, 1976; Shneiderman & McKay, 1976; Youngs, 1974), determine debugging strategies (Katz & Anderson, 1987; Murphy et al., 2008; Perkins & Martin, 1986), compare novice debuggers to expert debuggers (Ahmadzadeh et al., 2005; Gugerty & Olson, 1986; Jeffries, 1982; Perkins et al., 1986; Vessey, 1985), and examine how to teach debugging as an explicit process or skill (Carver & Risinger, 1987; Chmiel & Loui, 2003; Michaeli & Romeike, 2019b; Rich et al., 2019; Wescourt & Hemphill, 1978). However, studies that involved teaching debugging to upper-elementary and middle-school aged children (Carver & Risinger, 1987; Ko et al., 2019; Rich et al., 2019) did not investigate and enumerate students' existing debugging strategies so they could correct, extend, or augment them. Further, recent studies have focused on block-based programming rather than text-based programming, and so avoided debugging challenges associated with proper syntax. The current research aims to identify the debugging strategies and challenges that upper elementary-school aged novice programmers demonstrate and the effects of providing them with a set of explicit debugging strategies within the context of 8-hours of instruction during the course of an eight-week Python programming course. This chapter will introduce the study by first providing background on the problem of teaching debugging to novice programmers, followed by the specific aims of this study, the questions it addresses, its significance and limitations.

1.1. Study Background

Computer programming is challenging because it requires translating from a problem domain into the computing domain by writing out specific commands in a programming language. Any mistakes that the programmer makes in realizing this computerized version of the problem can result in either immediately reported errors or unexpected outcomes called bugs. These mistakes, missteps, or slips (Brown &

VanLehn, 1980) require the programmer to identify, locate, and repair these errors or bugs in a process called debugging. Expert programmers approach the debugging process using their knowledge of the programming language, their experience with similar bugs, and a set of self-developed heuristics for locating and fixing bugs they have seen before. Novice programmers on the other hand are still learning the language, have limited experience with similar bugs, and thus have little to no heuristics to use beyond what experiences they have from other domains. For the novice programmer, this process of acquiring the necessary skills to debug computer programs takes practice, experience, and persistence, or as one of my students exclaimed when asked to debug some code, "The code is like lava[1]," referring to the act of debugging as an uncertain and potentially perilous challenge.

This research aims to first identify what strategies novices come to debugging with initially and then evaluating the impact of teaching them explicit debugging techniques.

As an expert programmer with over forty years of programming experience, my own personal debugging skills developed through a combination of trial-and-error and persistence. I do not recall ever being taught or shown a formal debugging process and instead developed one on my own or copied techniques from others. The idea of novices developing their own debugging skills is something that early constructivists like Papert, Feurzeig, and Solomon (Feurzeig et al., 1969; Papert, 1972, 1980) suggested would happen but that Klahr & Carver (1988) reported was not common for the majority of novice students and teachers. As a teacher of computer programming for the past eight years, I have come to recognize the lack of understanding that students exhibit when faced with a new debugging situation. Without a clear set of steps to follow, they resort to unguided guessing, asking (and waiting) for help, or at worst giving up because they do not know where to start or are overloaded and frustrated.

In my search for explicit debugging techniques and pedagogy for novices, I have uncovered a few issues. The first is that the concept of a "novice programmer" is not well defined or agreed upon. Next, computer languages and environments are constantly changing. Finally, much of the research on debugging with novices has been constrained or limited. These constraints and limitations will be discussed below.

---

[1] The title of this thesis begins with a statement made by a student during the pilot for this study.

Defining what constitutes a novice programmer is a problem because computer programming is undertaken as a course of study at different times by different students for different reasons. For example, some students start in elementary school with casual or informal play-based learning, a concept initially popularized in the 1980's by Seymour Papert with his book Mindstorms (Papert, 1980) and his co-developed Logo programming language. As a result, there are a few novice debugging studies where the novices are children or adolescents (Carver, 1986; Carver & Risinger, 1987; Klahr & Carver, 1988; Ko et al., 2019; Rich et al., 2019). However, the vast majority of research studies looking at novice debugging involve college and university students (Ahmadzadeh et al., 2005; Böttcher et al., 2016; Chmiel & Loui, 2004; Jeffries, 1982; Katz & Anderson, 1987; Kessler & Anderson, 1986; Michaeli & Romeike, 2019b; Nanja & Cook, 1987). In other research, even adult *professional and scientific* programmers were also termed novices during early studies on debugging (Boies & Gould, 1974; Gould, 1975; Gould & Drongowski, 1974; Vessey, 1985; Youngs, 1974). As a result, when the literature discusses approaches or strategies for novices, the context of the study is critical. The approaches and techniques exhibited by a 'novice' adult professional programmer will be different from a child that is just learning other basic skills like reading, writing, and mathematics, or a university student looking to computing as a career.

In addition to the concept of a novice programmer being amorphous, the literature on debugging spans 50 years. As a result, the computer languages and environments have changed dramatically over this time. For example, many of the early studies were conducted "offline" using printed copies of the programs to be debugged[2] (Gould, 1975; Jeffries, 1982; Vessey, 1985; Youngs, 1974). It was not until the mid-1980's that studies switched to "online" debugging (Carver & Risinger, 1987; Gugerty & Olson, 1986; Katz & Anderson, 1987; Nanja & Cook, 1987) often using the then new personal computers and tools of the day. In addition to the computing devices having changed, the computer languages have also changed over time. Early studies used COBOL, Fortran, PL/1, and Algol. Moving into the 1980's, BASIC, Logo, Pascal, and Lisp were used in studies of novice programming. In the mid-1990's and onwards, Java was standardized for many undergraduate programs in computing science and so it

[2] Gould's studies did offer participants the option of using online tools, but most did not use them and instead stayed with the printed copies.

came to dominate many of the studies of novice programmers in the past 25 years. Meanwhile, though BASIC and Logo dominated earlier studies involving novice child programmers, much research has shifted to block-based programming environments like Scratch starting in the mid-2000's. Python, the programming language used in this study, was developed as a teaching language in the 1990's and has started to gain ground in high school and early undergraduate programs. However, there have so far been limited studies looking at Python novices and debugging specifically (D'souza et al., 2019; So & Kim, 2018; Whalley et al., 2023). In addition, the present study is the only one known to the author that has been conducted online via the online video-conferencing platform Zoom.

Finally, throughout the history of the debugging literature, like most topics of inquiry, there have been a number of constraints that have limited the studies in often practical but nonetheless impactful ways, relevant to this study. For example, many of the studies have involved primarily quantitative data collection and analysis, focusing on uncovering the frequency of errors and debugging approaches (Ahmadzadeh et al., 2005; Chmiel & Loui, 2004; Ripley & Druseikis, 1978; Spohrer & Soloway, 1986; Youngs, 1974). As well, timed studies have been common, namely, treating debugging time as the dependent variable (Gould, 1975; Gould & Drongowski, 1974) or limiting the time students are given to complete the debugging challenges (Gould & Drongowski, 1974; Gugerty & Olson, 1986; Katz & Anderson, 1987). Although limiting debugging sessions may be necessary for practical reasons such as limited researcher time, it can lead to using easier problems or incomplete results due to some participants failing to complete the tasks within the time allowed.

As a result of the challenges associated with defining who novice programmers are, choosing a modern programming environment, and settling on a method of inquiry, for this study I have chosen tools and an approach that I feel are timely and important. I have chosen participants in the age group 10-12 because I have been teaching students in that age group for eight years. As a result, I have experience working with them and the ability to recruit them. I initially chose to conduct my study via Zoom because of the limitations of in-person activities during the COVID-19 pandemic, though this also created data collection opportunities that were useful for my study. I chose Python because it is a popular modern language, developed for teaching, and commonly used as the first text-based programming language for this age group. Finally, I chose a

qualitative think-aloud verbal protocol approach because I wanted to begin by observing what strategies and challenges novice students come to or develop initially while learning how to program, and then see what effects introducing them to an explicit debugging strategy would have.

1.2. Research Problem

As previously noted, programmers spend a considerable amount of their time debugging; and novice programmers, who are less experienced both in programming and debugging, spend much of their debugging time just trying to get unstuck (McCartney et al., 2007). As a result, a number of studies have tried to uncover what novices know and what approaches they take. In addition, a few studies have looked at ways of providing explicit instruction on debugging (Böttcher et al., 2016; Carver, 1986; Chmiel & Loui, 2003; Ko et al., 2019; Michaeli & Romeike, 2019b; Rich et al., 2019) with the hope of easing novices' debugging burden. However, of those studies that have involved introducing explicit debugging instruction, all of them have come at the instruction with preconceived models or materials. Although informed by the literature and experience, none of these studies have started from the student's perspective. As a result, the existing literature focuses on what experts think novices need in terms of explicit debugging instruction rather than specific attributes and approaches based on pre-existing student approaches. Although these existing explicit approaches may work for some students, most studies report little to no change as a result of training students on them (Carver, 1986; Ko et al., 2019) or do not report on the impact of these explicit interventions specifically (Böttcher et al., 2016; Chmiel & Loui, 2003; Michaeli & Romeike, 2019b; Rich et al., 2019).

1.3. Research Questions

In response to the research problem identified above, this study aims to identify what debugging behaviours and approaches novice programmers ages 10-12 exhibit prior to debugging instruction and what impact teaching them explicit debugging techniques has on their subsequent debugging activities. This focus has led to the following objectives and subsequent research questions.

Research Objectives:

1. To identify common debugging strategies used by novice Python programmers.
2. To evaluate the effectiveness of introducing explicit debugging techniques to novice Python programmers aged 10-12 during 8-hours of instruction over an eight-week course.
3. To identify challenges that novice programmers face when debugging.

These three objectives have led to the following four research questions:

RQ1: What strategies do novice elementary school students employ when debugging Python code?

RQ2: What challenges do novice students face when debugging?

RQ3: What effects does an explicit debugging strategy have on student debugging strategies?

RQ4: In what ways do students respond to explicit debugging instruction?

1.4. Research Significance

This study will contribute to the body of knowledge on debugging by surfacing and discussing strategies used by young novice programmers and what impact teaching them explicit debugging strategies has. This will help to develop student-centered debugging pedagogy that can help identify misconceptions and provide explicit interventions in an attempt to reduce the “long tortuous cognitive development of the novice programmer” (Corney et al., 2012).

1.5. Research Limitations

Despite the potential contributions of this study towards the knowledge of debugging, this study has several limitations which affect its generalizability and replicability. First, the participants were all previous students of the researcher and

recruited to the study via convenience sampling. As noted in Chapter 4, an email was sent to all of the researcher's prior students, and nine students volunteered to join the study. As a result, these students represent individuals who have already expressed and acted on an interest in computer programming, have already undertaken at least one programming course using Python, and were presumably interested in the study course outline of using turtle graphics and Python. Similar studies involving more students, less experienced students, or differently motivated students may detect evidence of different strategies than those exhibited by the participants found in this study.

Another limitation may be the length of the study (8-hours over eight weeks). While this may seem long in comparison to some other studies, I might have preferred an even longer study duration. Unfortunately, it was necessary to restrict the length of the study due to the considerable time involved in teaching the course, transcribing the Zoom recordings, and coding and analyzing the observational data.

Another potential limitation of the present study is the choice to use video recorded observational verbal protocol session data. In addition to the time required to transcribe and code the data, the encoding also required substantial researcher debugging experience and was somewhat subjective in nature. In an effort to limit this subjectiveness, a second coder was used on a portion of the data, and then Cohen's kappa calculations were performed on the results until the two coders converged towards agreement. Despite this effort to establish the trustworthiness of the interpretations provided, other researchers may interpret the observations differently and thus arrive at different definitions of the strategies students employed. This in turn could then impact the pedagogical suggestions made.

Finally, as part of the study, three debugging challenges were presented to the participants, with each one being progressively more challenging. These three challenges were designed to present challenges that matched the course material and to assess students' usage of the explicit debugging techniques demonstrated to them. In addition, these challenges attempted to overcome or avoid limitations present in previous studies in the literature. For example, the students were not told how many errors were present in the code they were to debug; the challenges were conducted within the same development environment as their course work; and the sessions were

not time capped. Although these restrictions did avoid some of the limitations of previous studies, they also introduced new challenges. For example, by not capping the time allowed for students to solve a challenge and increasing the number of errors beyond a single one, the students took more time (in some instances, considerably more time). This resulted in class time overruns, additional stress on the students, and even the inability to complete the testing of all students during the final classes. As a result, although only three challenges were presented, their expansion may pose a challenge to future researchers as well.

1.6. Organization of the Thesis

This chapter has aimed to explain the context and motivations for this study. Chapter 2 provides a chronological review of the literature on debugging, starting with definitions for key terms to aid non-programmers. Chapter 3 details the design of the class in which participants took part, from the teacher-researcher's pedagogical perspective, stepping week by week through the progression of the instruction and debugging assessments. Chapter 4 describes the methodological approaches involved in the research and provide additional detail on the participants and the research setting. Chapter 5 details the data collected, and Chapter 6 discusses their findings and limitations. The appendices include the materials used or collected during the course of the study to support both transparency of the study as well as to support future researchers who choose a similar path or approach.

Chapter 2.

Literature Review

This chapter presents a review of literature relevant to debugging, its characteristics, and the research methodologies used to investigate it. Its major purposes are to: (a) define what debugging and its processes are, (b) evaluate current research on debugging, and (c) select and justify the area to be addressed in this research.

2.1. Introduction

While teaching upper-elementary students computer programming, specifically, a text-based language named Python, I noticed that most students stopped as soon as they encountered an error with their programs. After seeing this behaviour repeatedly, I began to investigate whether direct instruction on debugging techniques could reduce it. For example, during our first "hello world[3]" class I would ask students to purposefully introduce errors into their working programs by changing a word or deleting a character and then running their code again. This would immediately result in an error and block them from running their code. I would then lead them through reading and interpreting the error messages they saw, modeling how to read the error messages and possibly act upon them. However, this did little to change their apparent helplessness, so I went in search of ideas and instructional materials to fill this gap. This literature review details part of my journey towards understanding what strategies novice students come to the table with, and what effects teaching them explicit debugging techniques and strategies may have.

In reviewing the literature, I was expecting to find instructional materials and research studies to back up best practices on how to teach debugging. Instead, what I discovered was that there were only a very few studies that looked at explicit instruction on debugging, that calls to integrate debugging into programming instruction have gone

---

[3] A traditional first computer program is called "hello world" because it simply prints the text "hello world" to the console. A first class is also called "hello world" because it is typically students' first introduction to the programming language.

largely unheeded, and that after fifty years of research into debugging there is still little consensus on the strategies programmers use, how they develop them, and what strategies they *should* use.

This chapter reviews what we know about bugs, debugging techniques, and methods for teaching debugging, with the goal of informing the development of materials and techniques that will be effective in teaching successful debugging practices, as well as developing students' sense of agency and control when debugging.

2.2. Definitions

In this section, I will define the key terms related to debugging, to provide context for readers who are unfamiliar with debugging or computer programming.

2.2.1. Debugging Terms

Computer programs are a series of specifically structured commands that a programmer gives to a computer so that it can achieve a particular task (Ko & Myers, 2005, p. 43). Unlike natural languages (those that are spoken by humans, such as English), computer programming languages are formal languages designed to express computations (Wentworth et al., 2012, p. 5). As a result, programming languages have strict rules about their syntax and structure which are verified by a program called a compiler[4]. The job of the compiler is to read a program written by a human, verify that it is valid, and then convert the commands into the internal structures the computer will use to execute the commands. The compiler does not run the program itself, instead it translates the program commands into the runnable steps the computer will use in order to run the program.

Within the context of computer programming, the process of programming typically takes place within a programming editor. These editors can be simple text editors or fully integrated development environments (IDEs), Figure 2.1 shows the PyCharm IDE used in this study. The role of the editor is to provide an interface where

---

[4] To simplify the discussion, compilation is used even for interpreted languages.

the programmer can input and edit their computer program, in other words, write their code[5].

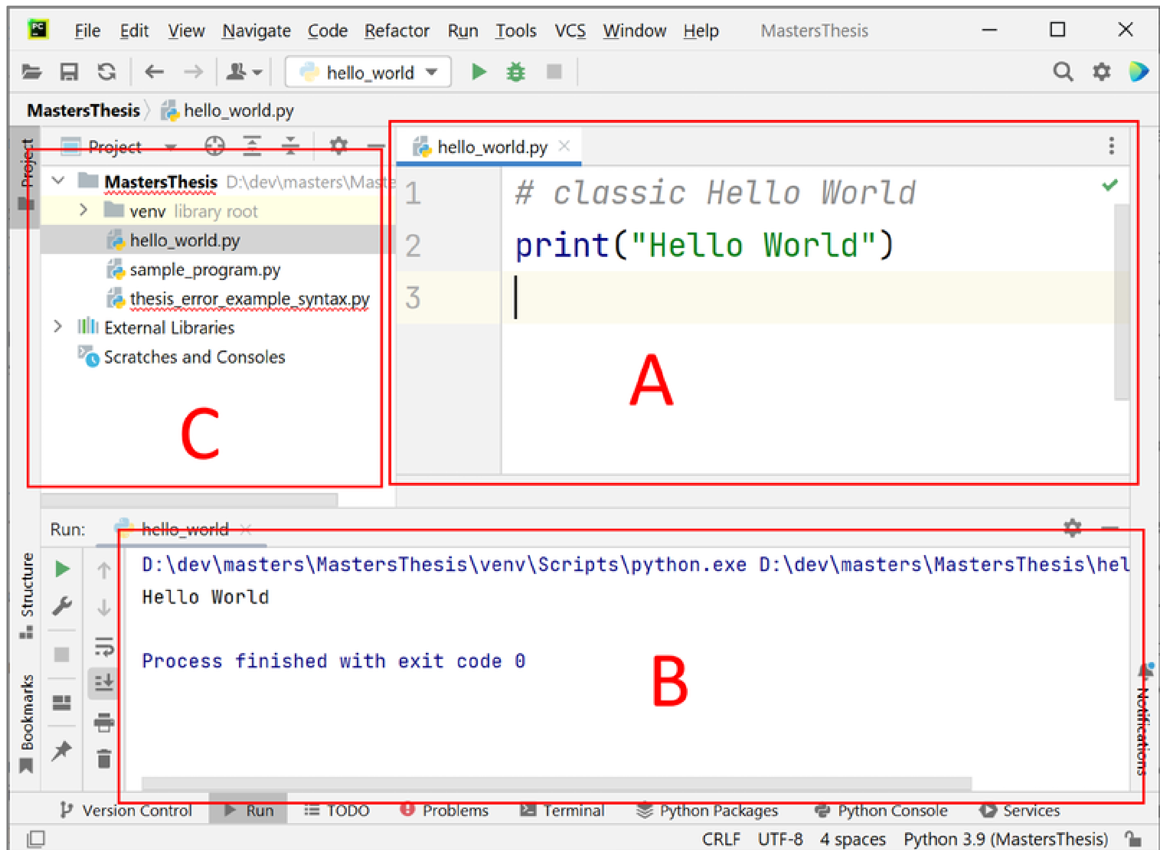


Figure 2.1. PyCharm Community Edition integrated development environment. (A) The code editing window for the script named “hello_world.py”. (B) The output window for program output as well as compiler or runtime error messages. (C) The project files, including the Python script files.

Once programmers have written their code, they submit it to the compiler whose job it is to parse the code and perform multiple checks on it. If errors are found during compilation, then the compiler will return a compiler error to the programmer with information about the type and location of the error (see Figure 2.2). Novice programmers will typically refer to this as their program failing to compile or run. If the program does not contain any compiler errors, then the program is ready to be run.

[5] This discussion will focus exclusively on text-based programming languages, which are typically written as words. Block-based languages operate fundamentally the same except they attempt to avoid written syntax to avoid syntax-related errors, but they are still susceptible to other errors.

```
1   # syntax error example
2   print("hello world")
3   print "I forgot the parentheses..."
4
```

```
  File "thesis_error_examples.py", line 3
    print "I forgot the parentheses..."
          ^
SyntaxError: Missing parentheses in call to 'print'.
Did you mean print("I forgot the parentheses...")?
```

Figure 2.2. Sample syntax error (missing parentheses) – Python 3.

In some environments, the program is run within the IDE and in others it is run separately; but in both cases the next step is to run the program. When the program is run, if an error occurs, then the program will exit and report a "runtime error". Novice programmers will typically refer to runtime errors as their program crashing. Runtime errors occur when the program attempts to take an invalid action that the programmer did not handle within their code, and so the computer aborts the program and reports where and why with a runtime error message. An example of a runtime error is when the program attempts to divide a number by zero, as this is a calculation whose result is undefined in mathematics (see Figure 2.3).

```
1   # sample runtime error - division by zero
2   x = 100
3   y = 0
4   divide_xy = x / y
```

```
Traceback (most recent call last):
  File "thesis_error_examples.py", line 4, in <module>
    divide_xy = x / y
ZeroDivisionError: division by zero
```

Figure 2.3. Sample runtime error (division by zero) – Python 3.
Code is syntactically valid, but crashes when run.

Finally, a program may be both syntactically correct and not violate any runtime rules, but still not correctly deliver or complete its assigned task according to an external set of requirements called a specification. A specification is a detailed description of the

target application that the programmer matches the output of their program with to determine if they have successfully created the target application. Errors involving a mismatch between the actual output and the specification are called semantic errors. Semantic errors occur when the program does what it was told to do rather than what was intended or required. For example, the program may draw a triangle (actual result) when it was supposed to draw a square (target image). See Figure 2.4.

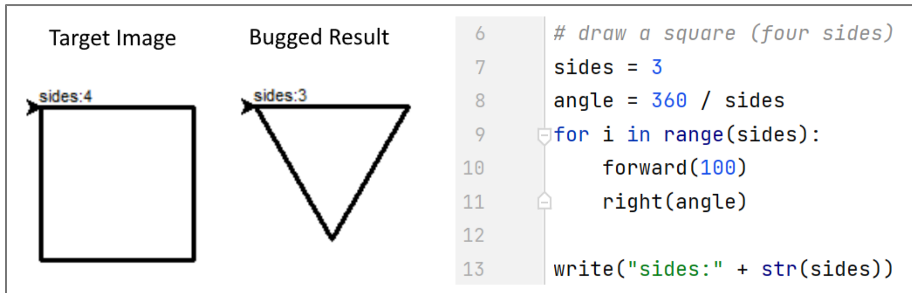


Figure 2.4. Sample semantic error – Python 3 Turtle Graphics.
Code is syntactically valid and does not violate any runtime rules, but its output does not match the target image called for in the specification.

All three types of errors (syntax errors, runtime errors, and semantic errors) are referred to as "bugs" and the acts of identifying, locating, and correcting them is known as debugging.

2.2.2. Debugging Processes

Early surveys of debugging concepts (Johnson, 1982; Kocher, 1969; Myers, 1978) defined debugging as a two-step process of locating and correcting errors that occurred in computer programs. However, this two-stage definition was insufficient because in order to locate and correct a bug, the programmer first needed to know that a bug existed. Therefore, a third step was needed, one of identifying that a bug existed. This resulted in a general three-step debugging process: identifying, locating, and repairing. This three-step approach will be used for this study.

According to Lukey (1980, p. 194) the first step when debugging, is to identify the bug by comparing the program's description to its specifications to arrive at a manifestation of a bug. Then, once a bug has been identified, the cause of the bug must be located. The second stage, locating, is described as reading through the code, making assertions about how it functions, back-tracking when a dead end is reached,

and repeating this searching until the bug is located. Finally, once located, step three is to edit or repair the bug. This results in a generalized three-step process of debugging by identifying, locating, and repairing.

Although the three-step process that Lukey described has been supported by others (Katz & Anderson, 1987; Ko & Myers, 2005), it has also been expanded and modified to suit the needs of individual studies. For example, Vessey (1985) and Klahr & Carver (1988) split the locating stage into two separate steps: gaining familiarity with the program and then exploring the program control and execution, resulting in a four-step process. Rich et al. (2019), when working with young children, proposed a different four step process of: observe, hypothesize, modify, and test. Carver (1986), also when working with young children, proposed a five-step process: run program, identify bug, represent program structure, locate bug, and correct bug. Carver's process is of particular interest in this thesis, because it was used for studying elementary-aged children and as such, it will be described in more detail next.

Of particular interest within the context of this study and to pedagogy related to working with novice child programmers, Carver (1986) expanded the original three-stage model into five steps (phases). As stated, Carver's first step is to run the program and compare the program plan or specification to the program's output. Although this new initial step is implicit in all of the previous multi-stage descriptions, novice programmers benefit from having an explicit step-by-step process that provides all of the steps. Also, bugs are typically not noticed until the program is compiled and run, so telling the students to run their program at the start and after each change is beneficial. Step two is to identify the bugs as discrepancies between the actual output and the intended target, just like the first step in the three-step process. Step three again makes explicit the implicit phase of representing the structure of the program by reading the code. This building of a representation of the program is necessary because it allows the programmer to create possible pathways through the code to follow and to reflect on how the program runs. Step four involves searching for and locating bugs. The last step is the correction of the bug once located. By explicitly stating the execution (step one) and representation (step three) elements of her five-phase model, Carver provided additional elements necessary for developing processes well suited to novices who require all of the steps to be spelled out explicitly. This stands in contrast to the needs of

experts who implicitly already know to run the program each time they make a change, and develop their own model of the program's code as they go.

2.2.3. Novices versus Experts

Throughout the history of debugging research, the approaches taken by expert programmers have been compared to those taken by novice programmers to identify differences. However, although the term "novice" is well understood to mean a beginner with little to no prior knowledge in the subject area, the term does not specify an age or what level of prior knowledge is the cut off between novices and experts. As a result, most of the debugging literature labels adult undergraduate students as novices because they are new to their field of study and thus assumed to have little prior programming knowledge.

A number of debugging studies have tried to address this by using different labels. For example, Kessler & Anderson (1986) labeled their participants as "true novices" because although they were adult undergraduates, they had no prior programming experience. Similarly, Miller (1974) used the term "naïve" rather than novice to describe participants with no prior programming experience. Nanja & Cook (1987) split their participants into three groups (expert, intermediate, and novice) based on their years of experience (graduate student, second-year, first-year). Ahmadzadeh et al. (2005) labeled their participants as good versus weak programmers based on their class grade, and then good versus weak debuggers based on their debugging performance. However, Vessey (1984) presents the biggest problem when talking about novices and experts, because she compared two sets of *professional programmers* with the goal of creating a measure of whether a participant should be considered a novice or an expert debugger based on their strategy and approach. Thus, she labeled less-experienced professional programmers as novice debuggers.

This thesis looks at novice child programmers, ages 10-12, who are both novice programmers *and* novice learners because of their age.

2.2.4. Debugging Code Written by Others

One of the challenges that researchers face when trying to investigate debugging is whether to give participants code the researchers wrote that contains bugs, or to ask the participants to write code themselves and observe them debugging their own code. Most studies choose to provide buggy code written by the researchers (code written by others) because this ensures that all of the participants are attempting to address the same bugs in the same code. This allows the researchers to control both the number and type of bugs for the purposes of the research. However, one consequence of this approach is that the participants must spend more time up front attempting to read and understand the code provided by the researchers before they can start debugging it. A second consequence of this approach is that the researcher-generated code doesn't necessarily contain the types of bugs that students would make in their own programs.

Many studies have noted this challenge and observed specifically that novices were slower than experts during the code comprehension phase (Gugerty & Olson, 1986; Jeffries, 1982; Kessler & Anderson, 1986; Lukey, 1980; Murphy et al., 2008; Nanja & Cook, 1987; Vessey, 1984). In other words, participants need to spend time reading and trying to understand code they did not write themselves. In contrast, a few studies have attempted to observe students debugging their own programs, and as a result focused on either documenting the different types of errors to inform language design (Boies & Gould, 1974; Litecky & Davis, 1976; Youngs, 1974) or counted the frequency of error types made (Ahmadzadeh et al., 2005; Ripley & Druseikis, 1978) to describe student actions in general. Two exceptions exist though. One exception is Katz & Anderson (1987), who used a mix of pre-written programs and participant-written programs. However, their study used an unusual editor, a LISP tutor, which resulted in the participants essentially filling in the blanks and thus guessing rather than performing more traditional debugging. The other exception is Murphy et al. (2008) who had students write their own version of a problem first and then gave them a buggy version of the same problem written by the researchers. This second exception was an explicit attempt to work around this problem.

In this thesis, I chose to use buggy programs that I had written, to ensure that all participants attempted to debug the same code. However, each of the debugging challenges were similar to the code the participants had already written during the

previous class(es) so that the focus was on debugging similar code. In addition, the study involved a series of debugging challenges, and so I was able to change the types of bugs for each challenge, in an effort to see how participants handled different quantities and types of bugs.

2.2.5. Turtle Graphics

A major part of this study relies on the use of a programming library named turtle graphics that allows for the creation of images using drawing commands that move a cursor calls a “turtle” around a digital on-screen canvas. (See examples in Figure 2.5.) Turtle graphics was created in the late 1960’s as part of the development of the Logo[6] programming language by Seymour Papert, Cynthia Solomon, and Wallace “Wally” Feurzeig. Logo was created to introduce mathematics to children using computer programming (Feurzeig et al., 1969). The original turtle (Papert & Solomon, 1971) was a robot with pens mounted in it to draw on paper; but future iterations, initially called “display turtles”, would display images on the computer screen instead. Eventually the “display” moniker was dropped and the process of drawing on the screen with a turtle cursor came to be known as turtle graphics. In addition, the physical robot would eventually be replaced by turtle graphics because it not only allowed more students to participate (due to the lack of reliance on scarce physical robots), but also because students could create animations by redrawing quickly on the screen.

---

[6] Logo is a name and not an acronym or abbreviation and thus is capitalized as such. However, in some studies it is capitalized as LOGO, in line with BASIC, FORTRAN, and COBOL.

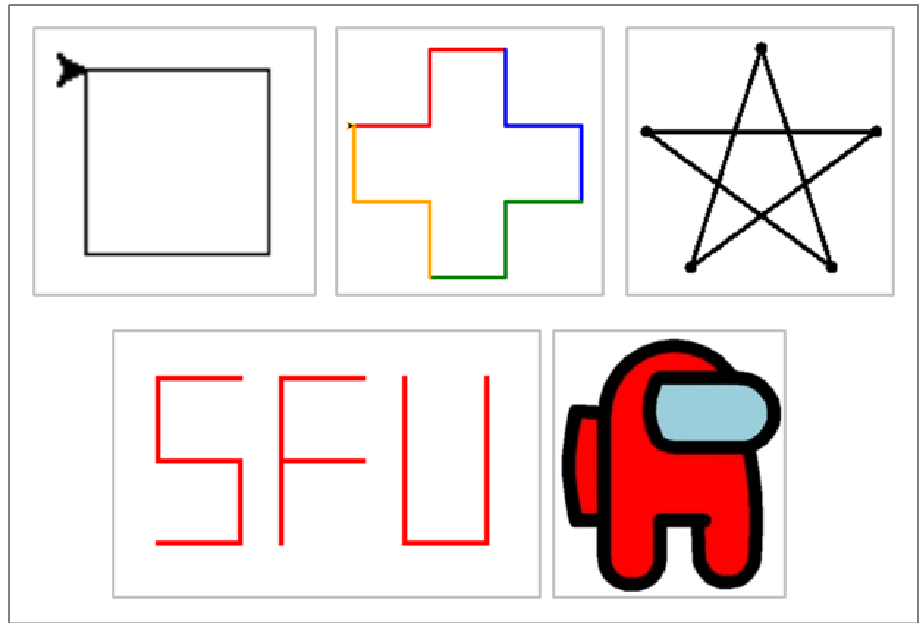


Figure 2.5. Sample images generated during the course using turtle graphics.

The reason that turtle graphics was chosen for many programming and debugging studies with children was that it provided an accessible graphical output that participants could recreate physically through a process described as “body syntonic” (Papert, 1980). In other words, students could understand what code did by “playing turtle” and enacting physically, using their bodies, the commands the turtle was programmed to follow. As a result, important concepts of programming and debugging could be explored both physical and graphically both with programs written by the researchers and the students.

Ultimately, Papert positioned teaching children programming (Papert, 1972) as an act of developing their understanding of concepts like mathematics through a process of “addition, refinements, and debugging” (p. 3.2). Later in his 1980’s book Mindstorms (Papert, 1980), Papert would argue that debugging was an essential component of both education and programming, especially for children because it established a mindset that errors (bugs) happen naturally, and that fixing them is a natural part of learning. This sentiment is echoed by Dweck & Elliott’s (1983) ideas about student motivation during learning, whereby mistakes (bugs) should be framed as learning opportunities rather than reflections of poor performance or a lack of intelligence.

2.2.6. Verbal Protocol Analysis

In order to provide context for the research methods used for debugging-related studies, including the present study, a short description of verbal protocol analysis is provided here with the other definitions.

In 1980, K. Anders Ericsson and Herbert Simon proposed that verbal reports could be used as data in studies of human cognition (Ericsson & Simon, 1980). As noted by Bowles (2010), researchers needed a way of determining the reasoning behind learners' actions, and verbal reports provided access to it. Without verbal reporting, researchers were left to infer what learners were thinking based solely on their actions and productions. With verbal reporting, it was argued, participants could verbalize their reasoning and provide access to their underlying cognitive processes. This approach is especially important for debugging research because it provides context for why participants took the actions they did, allowing researchers to note what features, assumptions, and knowledge they used for each of their problem-solving steps.

There are two primary forms of verbal protocol (Ericsson & Simon, 1993, p. 15): concurrent verbal reports and retrospective reports. In concurrent reports, participants speak aloud while engaged in the activity under study. In retrospective reports, participants are asked to provide an explanation of their actions after completing the activity. The advantage of concurrent reports is that the participants report their thoughts in the moment (in situ). The advantage of retrospective reporting is that the participants are not burdened during the task and can think about their answers after the fact. The disadvantages of concurrent reports are that the participant is burdened by the need to speak aloud while they are engaged in a potentially already challenging task, which may slow them down on the primary task and alter the cognitive processes involved with completing the primary task. The disadvantage of retrospective reports is that they are seen as less reliable because participants may forget, rationalize, or even justify their actions based on the outcome rather than describe the events as they actually happened. Ultimately, the choice of concurrent verbal reporting or retrospective verbal reporting comes down to the level of detail the researcher is looking to generate as well as the level of effort involved by the researcher and the participants.

It must be noted that, verbal reports are not without criticism. As noted by Bowles, asking participants to think-aloud while they are completing an already challenging task adds another task which may both slow them down and alter their cognitive processes and thus not result in the true insights claimed. Ericsson & Simon noted that participants should be given a warm-up or practice in the verbal reporting process before conducting the main observations, so they are aware of what they need to do and how to do it. As to validity, Nisbett & Wilson (1977) question whether participants can even access their unconscious processes explicitly during a task, and suggest that instead their reports are in-the-moment justifications for their actions rather than the actual processes used. In addition, protocol analysis is resource-intensive because participants' actions and words must be captured in the moment and then coded for interpretation after the fact. As a result, researchers must employ reliability measures, such as inter-coder reliability techniques, to ensure the trustworthiness of the research findings.

This thesis employed a concurrent verbal think-aloud protocol during the one-on-one debugging challenges to try to capture the moment-by-moment actions of the participants. In an attempt to alleviate the load of thinking aloud while debugging, early tasks were designed to be less mentally taxing. As well, the researcher provided instructions and practice on the process of thinking aloud, and reminded students regularly while engaged in the challenge tasks.

2.3. History of Debugging Research

Now that debugging and terms related to this thesis have been defined, I will discuss research on debugging related to this study. My aim is to provide a broad view of the research, and provide context for the decisions made in the design of the research for this thesis.

2.3.1. A Call to Teach Debugging

The first calls to teach debugging as part of computer programming instruction came from two separate groups of computer science professors. First, Heilman & Ashby (1971) called for the inclusion of debugging as an explicit topic in computer science education at the college level. This included a call for research into the classification of

errors, to facilitate the creation of teaching materials necessary to help students identify and correct bugs. Heilman & Ashby further suggested that debugging aids be created and integrated into programming languages, operating systems, and even the hardware to support effective debugging. Next, Mathis (1974) took the agenda a step further by making the case that debugging should be realized not as a standalone unit, but instead that it should be integrated throughout introductory programming courses to expose and teach the inner workings of the computer and the program, what today Lowe (2019) would call a "debugging-first" approach. Mathis suggested using techniques such as the use of print statements to expose program state, in order to show how to effectively localize faults in code. Unfortunately, both Heilman & Ashby's and Mathis' calls have gone largely unheeded. Instead, debugging continues to be glossed over, ignored, or at best taught as a standalone lesson or unit.

2.3.2. Initial Debugging Research

Early debugging research in the 1970's started with a focus on ways of reducing the time required to debug. As a result, the early studies looked at the types of bugs introduced (Boies & Gould, 1974; Youngs, 1974) and how long it took participants to debug them (Gould, 1975; Gould & Drongowski, 1974).

Much of the early research focused on what type of bugs were being introduced into code. Boies & Gould (1974), looked at syntax errors using three programming languages[7] and concluded that syntax error "[did] not seem to be a major bottleneck in computer programming" (p. 255), as they accounted for less than 1/6th of the errors observed. As a result, Gould's next studies measured how long it took programmers to debug non-syntactical errors. In two studies (Gould, 1975; Gould & Drongowski, 1974), participants were given program print-outs, told there was a single bug, and then timed as they tried to locate and report the bug to the researcher. These two early papers are among the most cited debugging-related papers, because they put forward claims such as that finding the bug (fault localization) is the most difficult of the three debugging processes (identify, locate, repair); that certain types of bugs take longer to detect; and that programmers are more efficient when debugging code they have previous experience with.

---

[7] Boies & Gould's (1974) three languages: Assembler, FORTRAN, and PL/1

Like Boies & Gould (1974), Youngs (1974) looked at where programmers made the most errors across five programming languages[8], measured where errors happened, described them quantitatively, and then reported which language constructs were the most error-prone. In addition, Youngs' study provided early examples of bug groupings and classifications, compared the differences between expert (experienced) and novice (beginner) programmers, and had participants write and debug their own code rather than providing them with code written by the researchers. Youngs' bug groupings and classifications provided insights into common sources of errors for both novices and experts, but most importantly, where novices differed from experts. In addition, having participants write and debug their own code resulted in naturalistic observations of the differences between novices and experts, which confirmed that their processes were different as well. For example, Youngs noticed that the experts were faster and focused on eliminating the easier syntax errors first before focusing on the semantic and logic errors. Conversely, novices made more errors, were not systematic when debugging, and took longer to debug. Youngs' observations about novices being unsystematic implicitly suggested teaching novices a systematic process to help them become more expert-like.

2.3.3. Criticism of Early Research Methods

At the start of the 1980's, researchers (Brooks, 1980; Sheil, 1981) advocated for rethinking how debugging and programming research studies were conducted. They claimed that many of the earlier studies were too focused on trying to measure within-subject variance.

Brooks (1980) raised three methodological concerns about the studies discussed above: the selection of measures used, the selection of participants, and the selection of materials used. Starting with the selection of measures, most early studies involved measuring how long it took individual participants to complete debugging tasks. However, because debugging involves a number of different sub-tasks, the question of when to time and what to time was not clear. In addition to what to measure, the question of who to measure (or the selection of participants) was another issue. Many of the studies noted wide variances between participants, but unfortunately did not attempt

[8] Youngs' (1974) five languages: ALGOL, BASIC, COBOL, FORTRAN, and PL/1

to account for this through other measures. For example, an expert professional programmer should be expected to complete debugging tasks faster than novice programmers just starting out. However, when measuring just the time to complete a task, there could be substantial differences between two experts or two novices. Thus, additional measures were necessary, for example measures of comprehension, skill, or knowledge. Finally, unlike many other tasks, two different debugging tasks may require completely different approaches; and so, the selection of the materials is challenging. As Brooks noted, many of the early studies tasked their participants to find and fix a single bug, with some even telling them that there was only a single bug (Gould & Drongowski, 1974)[9]. However, the level of difficulty, whether related to the number of bugs or the types of bugs, must be chosen according to the level of skill of the participants. Otherwise, the debugging task will be too easy and be completed too fast (resulting in a ceiling effect), or it will be too difficult, take too long and result in incomplete or ambiguous data. As a result of these three issues, Brooks suggested shifting the focus from trying to draw conclusions and make generalizations using only direct observations, and instead shift the focus towards the development of theories or models of the cognitive processes involved in programming.

Sheil (1981) added to Brook's suggestion of shifting towards behavioural studies because, he argued, programming is a complex human process, and many of the studies which stated that they were just trying to make something easier, faster, or less error-prone, were actually in fact psychological studies of cognitive processes. However, according to Sheil, this was problematic because many psychological theories are suggestive by nature and thus lacked the robustness and precision required to yield generalizable predictions or results. In other words, in a field like computer science that is based on logic and measurable outcomes, there was an expectation that experiments should be controlled experiments that produce generalizable results for the real world.

---

[9] To their credit, Gould & Drongowski (1974) did note that "[d]ebugging times were undoubtedly affected by participants' knowledge that there was always one and only one bug present" (p. 263) and suggested that future studies should look to vary the number of bugs.

Despite this, what seemed straightforward to measure (debugging), often failed to account for things such as practice effects, where repeating the same or a similar task multiple times could lead to improvements that could then result in quicker task completions on second and subsequent attempts. This was problematic, because the effects were methodologically weak but were presented as if they established strong claims. Sheil noted, surprisingly, that the computing community at the time paid little to no attention to how unclear the effects were in these studies, and instead presented them as strongly supported ideas around concepts of debugging, regardless of the fact that the sample sizes and effects were small, and the methods were weak. The criticisms did not go unnoticed, and signaled a shift towards trying to uncover the psychological and cognitive factors at play when debugging.

2.3.4. Verbal Reporting as Data

Fortunately, as the interest in investigating and formulating theories related to cognitive processes in programming and debugging began to form, psychology and cognitive science set about formalizing techniques for turning verbal reports into data, largely through the initiative of Ericsson & Simon (1980). With specific reference to debugging, at least seven studies during the 1980's (Carver, 1986; Gugerty & Olson, 1986; Jeffries, 1982; Katz & Anderson, 1987; Kessler & Anderson, 1986; Nanja, 1988; Nanja & Cook, 1987; Vessey, 1984, 1985, 1986) and a few more recently (Fitzgerald et al., 2005; Lewis, 2012; Murphy et al., 2008; Whalley et al., 2023) have looked at debugging using a verbal think-aloud protocol. As noted by Carver, the data collected needed to include the intermediate steps participants used in order to reveal the paths they took to solve a problem. As a result, debugging researchers used a think-aloud protocol to "solicit verbal expression of the knowledge currently active in the participants' short-term memory" (Carver, 1986, p. 41). This sentiment is echoed in the other studies cited above and forms the basis for developing traces of the steps involved with debugging so that participants' strategies could be more *fully documented* and then reported.

One of the first debugging studies to use protocol analysis was Jeffries (1982). In her study, four first-year undergraduate students (novices) and six graduate students (experts), were asked to report their thinking while attempting to debug printed copies of two Pascal programs that contained multiple bugs. Participants were told to report

errors as they discovered them, which allowed for the creation of a trace of their actions. However, although the experts presented their processes clearly, novices did not. This study intentionally made use of multiple bugs after noting the limitation in Gould & Drongowski (1974), which led to the discovery that a number of participants did not find all of the bugs. Novices missed 21% of the bugs and experts missed 8%. As well, Jeffries noted that novices read the code sequentially and introduced new errors while debugging, whereas experts read the code in the order it was executed, and rarely introduced new errors (p. 14). Finally, she noted that "All of these observations can be seen as evidence that novices were working under a severe memory load." (Jeffries, 1982, p. 10). Jeffries' study set the stage for more debugging studies using protocol analysis, novices versus experts, multiple bugs, and further investigations into the debugging strategies programmers of varying ability used.

Vessey's thesis (1984) is an early example of the use of a concurrent verbal think-aloud protocol during the debugging process in order to capture participants' raw thought processes as they debugged. Vessey states that this approach was taken so she could investigate the psychological processes underlying participants' debugging strategies. Vessey's study looked at sixteen professional adult programmers from a single company, who had been labelled as a novice or an expert debugger by their manager. The study presented the participants with increasingly difficult debugging challenges. Vessey recorded their actions and spoken thoughts, used these to create debugging episodes, and then used the episodes to classify each participant as a novice or an expert debugger based on the strategies they used. Her hypothesis was that differences between expert and novice programmers would become more apparent as the task difficulty increased, and so different levels of difficulty were incorporated into the study. Vessey's study is often cited as an expert versus novice study, but I feel this is a mischaracterization because both groups were professional programmers and the distinction between them was first assigned by their manager and then benchmarked by Vessey based on the strategies she determined from their debugging episodes. Vessey's study did, however, help to establish concurrent think-aloud verbal protocol as a viable, albeit resource-intensive (p. 427), means of observing debugging behaviours. As well, Vessey established the idea of using differences in debugging strategy between individuals as a way of classifying them as novice or expert. However, her study was not of novice programmers, but instead professional programmers labelled as either

novice or expert *debuggers* based on their demonstrated debugging strategies. As a result, additional research with actual novice programmers was needed.

Gugerty & Olson (1986) employed a concurrent verbal think-aloud protocol in their study which compared novice university students, who had just finished their first or second course in Pascal with graduate students labeled as experts. Their design was novel in that both groups knew Pascal, but were asked to debug both a Pascal program which they had some experience with, as well as a Logo program, which they had no prior experience with. Each program contained only a single bug, but participants were not told how many bugs the programs contained. Their study goal was to see if the debugging behaviours transferred to the new language. They found that experts were faster, introduced few new bugs or none, and appeared to comprehend the programs better. This led them to conclude that the expert's debugging advantage was as a result of their superior program comprehension abilities. They also noted that experts used "test writes"[10] (debugging print-statements) 50% of the time, while novices only 30% of the time. This suggests that both groups had some awareness of test writes as debugging aids, but not why there was a difference or if the difference shaped the outcomes of debugging.

An important contribution from Gugerty & Olson's study is a comparison of three debugging techniques: code comprehension, topographic search, and symptomatic search. They noted that experts had a greater knowledge of programming constructs, and this would likely lead them to understand what the program was doing more so than the novices. With regard to topographical search, where the participant used clues in the output or tests to narrow the possible locations of bugs, experts used this approach more effectively. As well, for symptomatic search, where the programmer used prior debugging experience and knowledge of similar past bugs to solve a problem, the experts could have a larger mental library of symptom-bug associations. Gugerty & Olson noted that debugging required using a combination of these three techniques to generate hypotheses for locating bugs, and that experts debugged more quickly and successfully than novices largely because they generated higher quality hypotheses and introduced fewer bugs while debugging.

[10] "Test writes" comes from Pascal's print function being named WriteLn rather than print like in Python and other languages.

Nanja & Cook's (Nanja, 1988; Nanja & Cook, 1987) study was an attempt to correct for a number of issues identified with previous studies. Specifically, their concurrent think-aloud verbal protocol study used eighteen participants classified as expert, intermediate, and novice based on their reported level of experience. The novices were first-year students, intermediates were third-year students, and the experts were graduate students who had worked in industry or taught the introductory programming courses. The buggy program they used contained multiple errors, and they measured seven attributes as shown in Table 2.1. Their key findings were that experts were faster, introduced no new errors, and employed a comprehension approach when finding bugs. The attributes described in Table 2.1 were used during the analysis for this thesis as a measure of the ability level of my novice students.

Table 2.1. Nanja & Cook's expert, intermediate, and novice attribute rubric

Attribute	Expert	Intermediate	Novice
Bug finding approach	employed a comprehension approach; understand the program first and then use that knowledge to find the bugs	employed an isolation approach; used output for clues to identify candidate bug locations, recalled similar bugs, and tested program state.	employed a random non-systematic approach; often characterized by trial-and-error or guess-and-check.
Bug correction approach	corrected multiple errors before verifying the corrections	corrected and verified single errors	corrected and verified single errors
Bug correction priority/order	corrected both semantic and logic errors at the same time	corrected the semantic errors first and then the logic errors	corrected the semantic errors first and then the logic errors
Bug correction speed	fastest	average	slowest
Bug correction accuracy	were most successful in correcting all of the errors	corrected all of the errors	did not correct all of the errors
Statement modification frequency	modified fewer statements.	made considerable modifications.	made very extensive modifications.
Introduction of new errors	did not introduce more errors	introduced several new errors	introduced many new errors

* Table data is based on attributes described in Nanja & Cook (1987)

2.3.5. Explicit Debugging Instruction with Children

There are a limited number of studies that address teaching children an explicit debugging process. David Klahr and Sharon Carver were the first to do this in the 1980's (Carver, 1986; Carver & Risinger, 1987; Klahr & Carver, 1988, 1988). Recently, a couple of studies have looked again at debugging with young children (Rich et al., 2019), and teenagers (Ko et al., 2019; Michaeli & Romeike, 2019b). Carver's work will be the initial focus of this review, as it was the original model that was followed for the design of this study, while Rich et al. and Ko et al. provide some modern perspective, and Michaeli & Romeike furnish insights from high school teachers as well as students.

Carver and Klahr conducted a number of debugging-related studies with elementary-aged students. Their initial pilot study (Carver & Klahr, 1986) looked at the

effect of students engaged in "guided discovery learning" using Logo and turtle graphics. According to Carver, "A guided discovery environment is one in which the instructor introduces new LOGO concepts and may give project ideas for trying them, but then the students are free to create projects of their own choice." (p. 23). This guided discovery model describes the type of learning environment that Papert advocated (Papert, 1972, 1980), which he claimed promoted deep learning and transfer of skills.

In addition to the guided discovery environment, students in Carver & Klahr's pilot study were given explicit debugging instruction and a specific protocol to follow, as shown in Figure 2.6. Their approach was successful in showing that teaching students an explicit debugging process helped them with their programming performance and confidence. However, this contradicted Papert's claim (Papert, 1980) that students would develop their own debugging skills through self-discovery. Instead, Carver & Klahr reported that in the absence of explicit instruction in debugging neither students nor teachers developed anything beyond "the most meager debugging skills" (Klahr & Carver, 1988, p. 377). They claimed that explicit debugging instruction was necessary for novice programmers to develop effective debugging techniques, and set about exploring the impact of such instruction. Following the results from her PhD thesis (1986), Carver conducted another study with Risinger (Carver & Risinger, 1987) where they took the explicit debugging instruction lessons learned from Carver's previous work with Klahr and used it with another group of students. The key findings from these studies were that explicit debugging instruction for novice child programmers is both necessary and effective.

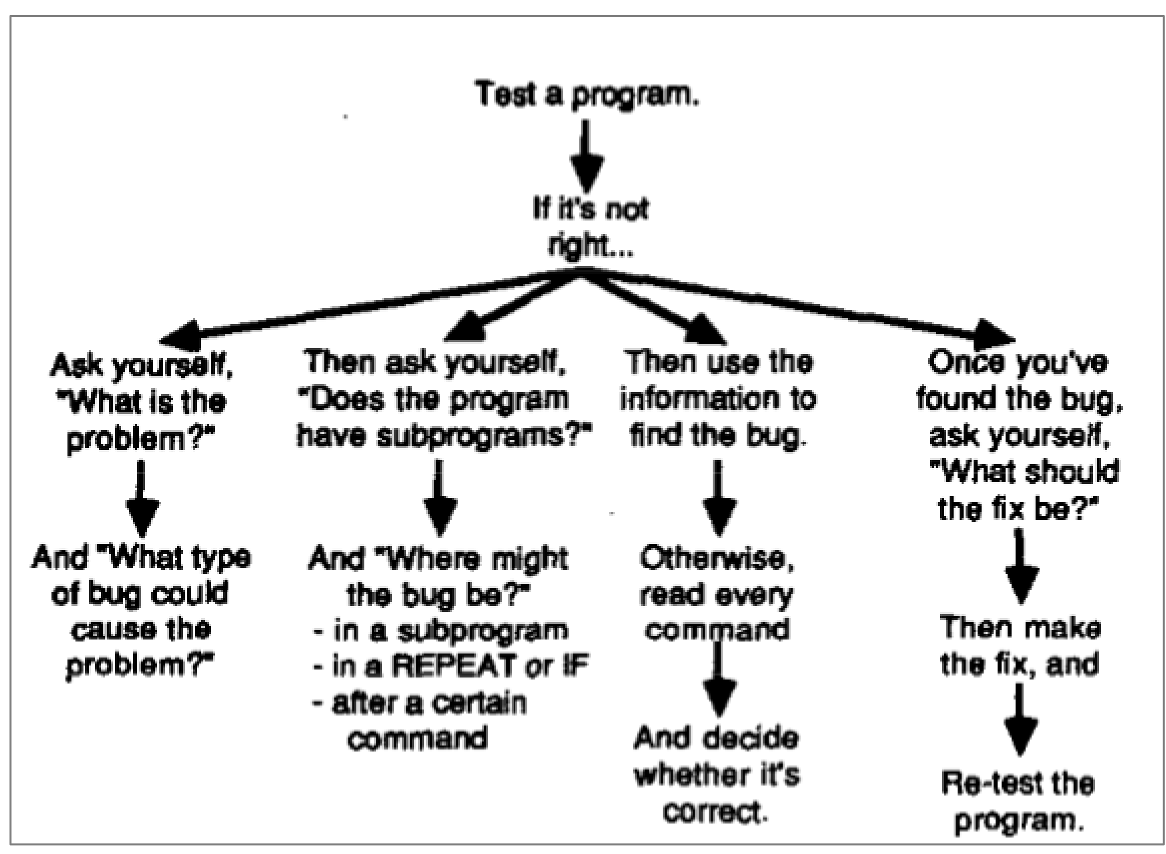


Figure 2.6. Debugging flowchart from Klahr & Carver (1988, p. 378).

Carver's PhD thesis (1986) played an outsized role in shaping this thesis. After reading Carver's thesis my original intent was to replicate it, however I was unable to obtain a complete copy of her 35-year-old thesis. Instead, I was only able to obtain the main body of the thesis, without the appendices which included many of the artifacts necessary for replication. The reason I was originally interested in replicating this study is that it surveyed a similar age of student to those I had already taught, used a think-aloud protocol, covered turtle graphics over a period of twelve weeks[11], and included not only debugging testing but also debugging instruction, i.e., this was a course with debugging testing. Undeterred by my inability to access Carver's thesis appendices, I modelled key parts of this thesis after her course outline but developed my own course materials, assessments, and changed elements I observed in other studies.

[11] Students were taught Logo for 6 months (24 weeks vs. my 8 weeks). There were two programs, one using turtle graphics and one using list processing. Each program was 12-weeks in length, each session was two hours in length, resulting in students receiving 2 hours/session x 12 sessions x 2 programs = 48 hours.

2.3.6. A Renewed Interest in Debugging

By the end of the 1980's research into debugging slowed, with only a small number of studies occurring at the start of the 1990's which summarized the work done in the 1980's (Gilmore, 1991), copied previous studies (Allwood & Björhag, 1990), and showcased yet another tutoring aid for programming (Araki et al., 1991).

It wasn't until ITiCSE[12] 2001 that interest in debugging re-ignited, after a working group led by McCracken (McCracken et al., 2001) reported that following completion of their first one or two programming courses, first-year undergraduate computer science students, exhibited a surprising lack of programming competency. A follow-up study by Lister et al. (2004), looked at code reading and tracing, and resulted in calls for research into the underlying issues that novice computer science undergraduate students faced. One result of these two working group reports was a qualitative study by Fitzgerald, Simon, & Thomas (Fitzgerald et al., 2005) that used concurrent verbal think-aloud as the basis for a grounded theory of tracing strategies of novice students.

This study's goal was to identify strategies such that they could be developed into theories of how students approached the reading and understanding of code. Although the study did not focus on debugging specifically, it was soon recognized that code tracing was a core technique used when trying to locate a bug during debugging. This led Simon, Fitzgerald, and others to form an international working group that consisted of multiple institutions in the United Stated and United Kingdom to review and revisit the debugging literature (then at the time at least twenty years old). This collaboration resulted in one literature review and four research studies. This set of studies, starting in 2008, focused on debugging and relied extensively on the work conducted in the 1980's. Similar to this thesis, the researchers made changes to account for the introduction of modern programming tools, languages, and environments. However, their focus was on first and second-year undergraduate students. A review of these 2008 studies follows.

In their review of the debugging literature, McCauley et al. (McCauley et al., 2008) focused on the teaching of debugging. They noted that the literature was rich but dated. In their findings, they observed that few introductory programming textbooks

---

[12] ITiCSE is the ACM conference on Innovation and Technology in Computer Science Education.

covered debugging, and those that did provided either basic definitions or just general rules of thumb to guide the novice. Their definition of debugging: “find out exactly where the error is and how to fix it” (McCauley et al., 2008, p. 68) covers the two general ideas of locating and fixing errors, but not the third of *identifying* nor Carver’s additional three, all of which are beneficial for novice programmers. They noted that bugs occurred because of language misconceptions and a chain of cognitive breakdowns (Ko & Myers, 2005) related to the rules of the language, student’s knowledge of the language, and student’s skills using the language. They also (importantly) noted that educators should not ignore the incorrect mental models that are the root cause of unsuccessful debugging, and should therefore teach correct debugging techniques. However, unfortunately, although their paper touched on a number of historical concepts and ideas, it did not provide any explicit strategies that practitioners could use within their classrooms, and instead provided a list of historical studies and concepts covered. Nonetheless, this literature review would act as a starting point for subsequent studies, that would look to revisit core concepts related to debugging: what debugging strategies novices used when debugging code, and what debugging looked like from a student’s perspective.

In two subsequent studies, Fitzgerald et al. (2008) and Murphy et al. (2008) reported on the debugging strategies used by novice undergraduate students while debugging Java programs. These studies used a combination of interviews, a programming exercise, and a debugging exercise. During the debugging exercises, they used a concurrent verbal think-aloud protocol. The first study (Fitzgerald et al., 2008), served as a confirmation of findings from previous studies, namely that once students find a bug they can fix it. This led them to suggest teaching tracing strategies derived from their previous study (Fitzgerald et al., 2005): tracing code mentally in their heads, on paper, using print statements, and via a debugger. This then led to a set of debugging heuristics to help novices decide on the best approach for a given situation. See Figure 2.7.

- *if you have to keep up with more than one or two variables or there's a loop involved then you need to trace on paper, not just in your head;*
- *if the bug can't be determined by looking at (sic) only at the input and output values then you need to add some print statements in the middle;*
- *make sure that your print statements are well placed and print useful information;*
- *if you have to put in too many print statements, your program 'hangs' or you think it has an infinite loop, use the debugger.*

Figure 2.7. Debugging Tracing Heuristics (Fitzgerald et al., 2008, p. 114).

Of importance to this thesis, Fitzgerald et al. noted a challenge while conducting the think-aloud portion of their one-on-one sessions with students; namely that students reported finding thinking aloud distracting, and often chose not to speak. As a result, the researchers reported limits to the conclusions they could draw from some of the participants' actions. This led them to recommend developing a protocol that encouraged more speaking, and also recording the actions of the participants to provide more context.

In their second study, Murphy et al. (Murphy et al., 2008) attempted to reduce the impact of code comprehension at the start of a debugging exercise with code written by others. To do this, they first asked students to choose one of six typical CS1 activities to code. The students were then given 30 minutes to program their own solution to the problem. Next, after the students had created their own version of the exercise, they were given a syntactically correct solution to the same problem, which contained 3-5 logic errors. Students were then given 20 minutes to debug this bugged version, while the researchers maintained a minute-by-minute log of their activities. The rationale for first having participants create their own solution to the problem and then asking them to debug a bugged version was that the researchers wanted the students to be familiar with the problem, and thus bridge between their own code and the code written by others. In addition to the activity log files, the researchers also conducted semi-structured retroactive interviews to elicit participants' impressions of the debugging exercises, strategies used, and motivations for their problem choices. This study resulted in thirteen "good" effective strategies, nine "bad" less effective, faulty, or unproductive strategies, and a handful of "quirky" or unusual strategies (p. 165). Of particular interest, the researchers noted that when tracing, some students used debug print-statements

incorrectly, didn't appear to understand the code even though they edited it, commented out suspicious lines that were correct, and worked around problems by replacing them with new code in order to avoid fixing them.

While discussing the observed strategies, Murphy et al. noted that some students "tinkered" with the code. They described tinkering as "fairly random acts that usually resulted in unproductive changes" (Murphy et al., 2008, p. 166). They noted that tinkering was almost always ineffective, whereas systematic tracing was usually effective when used correctly. This implies that tinkering should be framed as an ineffective approach with students, and instead systematic tracing should be taught with explicit instruction and practice. They noted that despite a lack of explicit instruction, students seemed to be familiar with and used many common debugging techniques. However, they also noted that because of this lack of explicit instruction, many students applied the techniques incorrectly, inconsistently, or ineffectively.

In a related study in 2010, Fitzgerald et al. (Fitzgerald et al., 2010) looked at debugging from the students' perspective, by asking students to retroactively describe their debugging approaches. They noted that students relied heavily on online resources, using what they called "pattern matching" to try to find solutions to their problems. However, when students failed to find an answer online, they flailed because they lacked core debugging skills. Fitzgerald et al. noted that tracing skills were underdeveloped (something they noted previously in 2005 and 2008 as well), which led to a suggestion that tracing skills be emphasized as both a means of understanding the code but also a useful debugging technique. A key takeaway regarding debugging code written by others, as in this study, was offered by a student, who commented: "If I wrote it, I'm already intimately involved in the algorithms of the code…. If it's somebody else's, It's figuring out what the heck they are doing."(Fitzgerald et al., 2010, p. 394). As a result, the researchers suggested explicitly demonstrating techniques such as commenting out blocks of code, using a debugger, but also being mindful of the additional cognitive load required to learn and use the debugger within an already demanding context.

All in all, these 2008 and 2010 studies helped to revisit and refresh some of the issues and findings raised in older studies. The main points stressed and emphasized were the teaching of tracing, being mindful of cognitive load, and both the value and

challenges of verbal think-aloud protocols as a technique for studying the cognitive processes of novice programmers.

2.3.7. Debugging Instruction in the Classroom

Starting as far back as Heilman & Ashby (1971), a common call to action has been that debugging should be taught as part of instruction in computer programming. This has resulted in common retorts that either students will pick it up as they go (Papert, 1980), that textbooks already provide debugging instruction, or that teachers will provide just-in-time instruction as necessary. However, as noted by Carver & Klahr (1986) most students did *not* develop effective debugging processes without explicit instruction. As well, McCauley et al. (2008) noted that "typically little to no space is devoted to bugs and debugging in most introductory programming textbooks" (p. 68). Recognizing the lack of instructional materials available to support the teaching of debugging, three recent studies created and tested classroom materials (Böttcher et al., 2016; Ko et al., 2019; Michaeli & Romeike, 2019b).

The first of the group to test adding debugging instruction to a programming course was Böttcher et al. (2016). They created a debugging unit within their Java programming course for first-year university students. Their study looked at the effects of adding a single unit on debugging, which consisted of one reading, one lecture, one lab, and one assignment. The focus of their unit was on using Gauss' (1982) Wolf Fence algorithm[13] for fault localization, using the debugger integrated into their IDE, building and using unit tests to validate their code, and writing a report detailing how they debugged the assignment. The student assignment consisted of a single program that contained three bugs and was graded based primarily on the student's ability to effectively describe their debugging process. They noted that they had observed the following naïve debugging strategies: random visual inspection, and print statements to visualize process flow changes in program state. Results from the study showed that less than 10% of the students used the Wolf Fence approach, and that there was a weak correlation between students' written debugging reports and their debugging abilities. Overall, this study did not provide convincing evidence to support the approach taken;

[13] The Wolf Fence algorithm is a divide-and-conquer search algorithm that splits the search space in half repeatedly by testing a condition (a wolf howl) in one half and if not present then the wolf must be in the other half.

but this may be considered unsurprising since it was only a single unit and not integrated throughout the course as suggested by Mathis.

In 2019, Michaeli & Romeike (2019a, 2019b) published two papers on explicitly teaching debugging during undergraduate computer programming courses. Their first study (Michaeli & Romeike, 2019a) concluded that although they considered debugging an essential skill, it was not being explicitly taught in classrooms. Digging further into the issue, they interviewed German high school teachers to investigate how they approached debugging in their computer programming classes. Their results showed that weaker students were often seen as helpless and used unsystematic trial-and-error approaches. Also, they discovered that compile-time syntax errors played a much more significant role than is often reported in the debugging literature (Boies & Gould, 1974). This may in part be because this study looked at younger novice students in high school rather than in university; so, these students were still stuck on lower-level errors than those common among older students. Although teachers did report talking about heuristics for addressing common bugs, they did not have a systematic process to teach the students due to a lack of available materials.

Following their study with high school teachers, Michaeli & Romeike (2019b) developed and then introduced a three-level systematic debugging process focused on student self-efficacy (see Figure 2.8). They then conducted a pre-post control-group study to examine the efficacy of their debugging process. Both the experimental and control groups were surveyed using a questionnaire and given debugging exercises as a pre-test. Both groups completed debugging exercises, with the experimental group using the intervention and the control group just doing the debugging exercises. For the post-test, the students were surveyed again and also completed debugging exercises without any explicit preparation. Their results showed a significant increase in both self-efficacy expectations and debugging performance in the experimental group as compared to the control group.

Michaeli & Romeike claimed that their study provided empirical evidence for explicitly teaching debugging and provided a hands-on approach for the classroom. However, the intervention, debugging lesson, as well as the pre- and post-tests were all covered within a single 90-minute lesson taught by the researchers and accompanied by the poster shown in Figure 2.8. Furthermore, their explicit debugging intervention was

only 10 minutes in length (p. 3) and thus subject to strong recency effects. Also, it is not clear that the intervention was entirely responsible for the observed differences between the experimental and control groups. Students were not randomly assigned to groups, individual student debugging proficiency was not measured before the intervention, and students' levels of programming performance were not accounted for. So, similar to Böttcher et al. previously, the intervention was limited, and other variables were not controlled for to support generalization of the results.

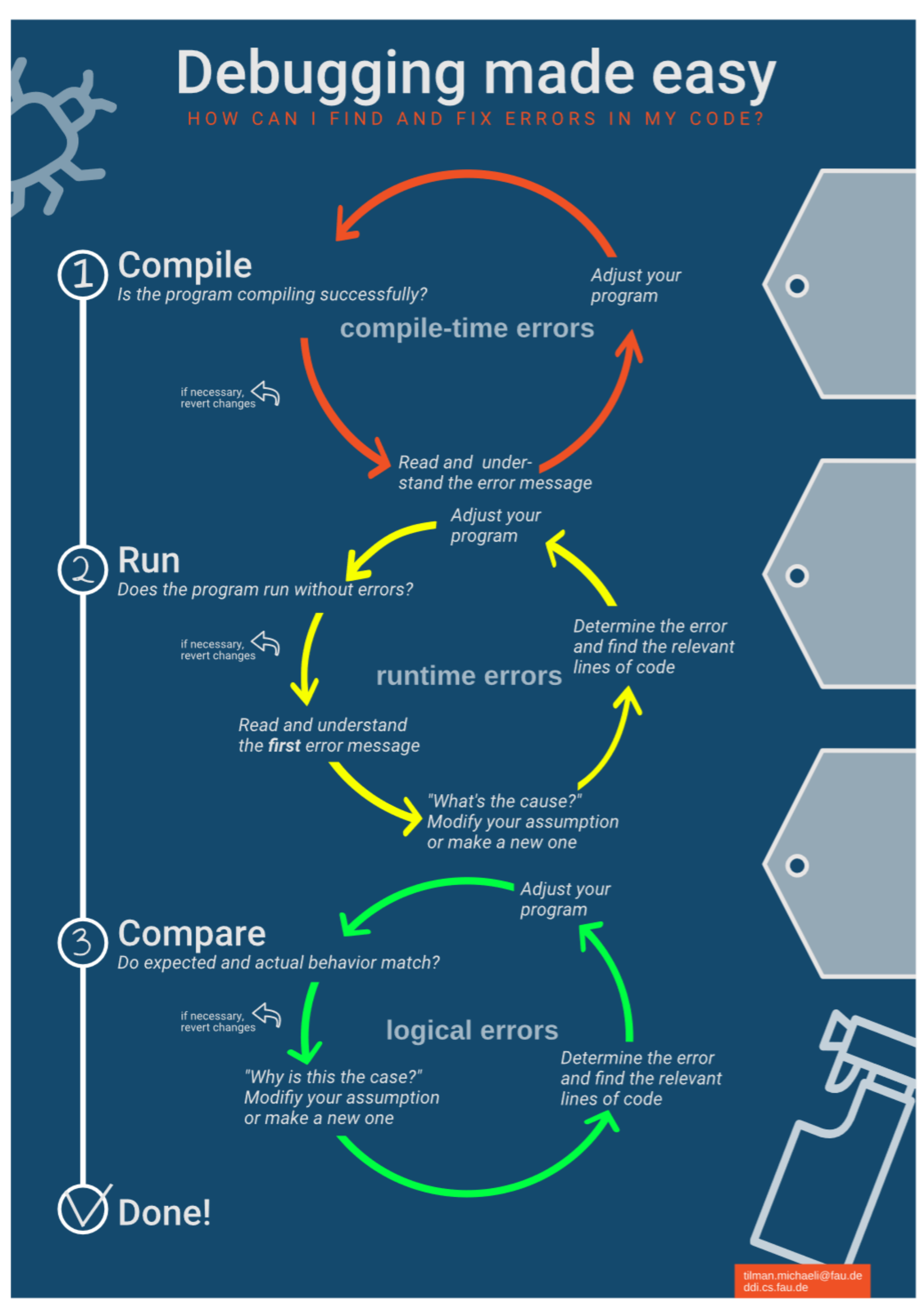


Figure 2.8. Michaeli & Romeike's three-level systematic debugging process.
From Michaeli & Romeike (2019b, p. 5), “Figure 4: Systematic debugging process conveyed in the intervention”.

In a study designed to investigate debugging strategy selection and self-regulation, Ko et al. (2019) introduced seventeen high-school students to two strategies for debugging and code reuse during a five-week summer camp that focused on video game development using JavaScript. The students were novices who had little to no prior programming experience. Students followed course material from Code.org for three weeks, and then spent the final two weeks building a game of their choice, alone or in pairs, over 10 classes. Explicit debugging strategies were presented as a series of generic conditional steps that students could follow. See Figure 2.9.

```
# If you need help finding the problem, ask for help.
Find what your program is doing that you do not want it to do
# Write the line number inside of the program
# and separate with commas.
SET 'possibleCauses' to any lines of the program that
    might be responsible for causing that incorrect 'behavior'
FOR EACH 'cause' IN 'possibleCauses'
    Navigate to 'cause'
    # Ask for help if you need guidance on how.
    Look at the code to verify if it causes the incorrect behavior
    IF 'cause' is the cause of the problem
        # If you need help finding the problem, ask for help.
        Find a way to stop 'cause' from happening
        # Ask for help if you need guidance on how.
        Change the program to stop the incorrect behavior
        # Ask for help if you need guidance on how.
        Mark the task as finished
        RETURN nothing
IF you did not find the cause
    Ask for help finding other possible causes
    Restart the strategy
RETURN nothing
```

Figure 2.9. Debugging strategy text (Ko et al., 2019, p. 471).

Ko et al. reported that although students said the strategies were valuable, many had trouble regulating their choice of strategies and instead defaulted to an ineffective trial-and-error approach. Those students who used the recommended strategies completed more features in their projects, but this association may have been mediated by factors other than the strategies alone. They concluded that students may require more explicit instruction on strategy selection and self-regulation. This observation aligns with observations from Fitzgerald et al. (2008), who noted that even with explicit strategies, students still defaulted to poor strategies and got stuck. It also echoes findings that students may ignore recommended strategies that they have been taught.

2.3.8. Debugging-First: A New Approach to Teaching Debugging

As just discussed, the teaching of debugging as part of computer programming is arguably logical, but subject to the constraints of limited time and resources in the classroom. A different perspective suggested by Lowe (2019) is that rather than considering debugging as a separate skill that one acquires, debugging should instead be integrated throughout introductory programming courses through a "debugging-first" pedagogy. In 2019, Lowe introduced a new theory of programming built upon dual process theory and Bruner's theory of representation named the Theory of Applied Mind of Programming (TAMP) (Lowe, 2019). Dual process theory (Frankish, 2010; Kahneman, 2013), refers to a model of the human mind based on the concept of it having two systems available for cognitive tasks, named System 1 and System 2. System 1 is often referred to as the fast-access, automatic, heuristic system that allows for immediate answers and actions, while System 2 is the slow-access, rule-based, analytical, or reflective system that allows humans to reason about problems thoughtfully. Bruner's theory of representation refers to a model where learners, when faced with new material, go through a three-stage process that connects this new knowledge with existing knowledge. Thus, according to Lowe, by engaging with programming using debugging, that debugging techniques move from relying exclusively on rules (and thus the slower System 2) to becoming automated like the faster System 1.

As a result, Lowe's theory suggests that rather than teaching debugging as a nice-to-have skill or an implicit skill developed as a result of programming, instead, debugging should be used as the central means of teaching programming itself. This challenges the notion that success in CS1 is appropriately measured through demonstrations of *knowledge* rather than demonstration of *skills* -- an assumption that McCracken et al. (McCracken et al., 2001) reported among teachers and which spawned Lister et al.'s 2004 ITiCSE working group, Simon et al.'s 2005 work, and then the 2008 and 2010 studies on debugging, as previously discussed.

Lowe's approach echoes Mathis' (Mathis, 1974) call for the integration of debugging into programming, but takes it a step further by suggesting the need for a "debugging-first" pedagogy that can perhaps fill some of the mental gaps that TAMP suggests hamper novice programmers. For instance, using worked examples, where

errors are introduced and then debugged using explicit steps as well as test cases, tracing, isolating the wrong line, or correcting individual lines of code. As a result of this approach, many debugging problems can use the same scaffolding already used during programming instruction by introducing new bugs along the way and then working to identify, locate, and repair them. In an effort to minimize cognitive load, Lowe suggests that the teaching "should stress continuous practice with basic materials to the point that they become overlearned" (p. 2). This concept of overlearning describes the automation of rules normally processed by System 2 towards the creation of System 1 processes, which are ultimately immediate and costless. This approach points towards using debugging from the start, integrating it into introductory programming, with the goal of automating the debugging process early and within context. As a result, Lowe suggests that the debugging-first approach should help address issues raised by McCracken et al. around students not knowing how to program (and debug) by the end of their computer science programs.

The present research did integrate explicit debugging instruction, as well as providing and working through debugging situations while teaching Python and turtle graphics. As a result, the approach taken in this research is similar to the approach advocated by Lowe. However, given the short duration of the study, with only 8 hours of total instructional time, any effects were preliminary. Therefore, generalizability is limited.

2.4. Literature Review Limitations

The following are some limitations of this literature review.

Age of the Research

The majority of research on debugging is from the 1970's and 1980's. This review uses older research to build up the context for the more contemporary research. Although the general concepts of computer programming remain the same over the time period discussed, computers and programming languages have changed considerably over this time. As a result, although the literature is old, it is still relevant and, in some cases, provides the only examples of research related specifically to debugging.

Limited Research on the Topic of Children and Debugging

The majority of the research on debugging has been conducted using undergraduate students rather than children. Of the many studies reviewed, only a handful were conducted with children and of those, only a few looked at explicit instruction on debugging. In addition, of those that looked at explicit instruction with children, only Carver & Klahr (1986) and most specifically Carver's thesis from 1986, looked at the impact of explicit debugging instruction on children.

Excluded Research

Two classes of recent research were not included in this literature review: debugging e-textiles and debugging block-based programs. Electronic textiles (e-textiles) involve physical creations, often controlled by electronics that are programmed, but after reviewing some of the recent studies, their focus was sufficiently different, especially as it relates to debugging, as to be considered not relevant to the present study. Similarly, block-based programming languages like Scratch, although mentioned in some of the included studies, are aside from the focus of this research, which is text-based programming.

2.5. Research Topic Justification

Using the materials reviewed and discussed in this literature review, the following is a justification of the design choices made for this thesis. In choosing and justifying the research topic for this thesis, I have chosen to return to Brooks' (1980) criticism of early debugging studies, in which he raised three methodological concerns: the selection of measures used, the selection of participants used, and the selection of materials used. Based on this literature review, I made the following choices. For measures, I chose to compare students' debugging strategies both between participants as well as over the course of the study. For participants, I chose upper elementary students (Grades 5-7, ages 10-12) because they are an under-studied population that is, nonetheless, increasingly a focus of instructional effort. Lastly, for materials I modified what I was able to glean from Carver's thesis (Carver, 1986), which focused on similar participants and was looking for similar effects of explicit debugging instruction. In addition, I modified the codes used in Vessey's thesis (Vessey, 1984) for coding the debugging session transcripts.

With respect to, measures, I chose not to focus exclusively on measuring debugging *times* because I felt that although a useful measure of relative performance, it is also fraught with issues of accuracy and subject to wide variances for reasons related to the task at hand and the problem presented. In addition, I chose not to focus on *frequency* of errors because I felt those failed to answer the question of why students made the choices they did, to arrive at the bugs being recorded. Although useful for spotting trends, the frequency of errors focuses too much on the output rather than the reasoning. Therefore, I chose to develop a measure of the strategies used so that I could investigate *how* students debug, and then see how their approaches changed after being introduced to explicit debugging instruction.

In order to develop the strategies to be taught, I needed a way for participants to both show and tell me their strategies, which led me to choose using a verbal protocol as well as screen recording. I ultimately decided to use a concurrent verbal think-aloud protocol over a retrospective verbal protocol, because I wanted to capture students' thoughts in the moment rather than after the fact. Although challenging and resource-intensive, I felt that by practicing the process during a pilot study beforehand, to give me experience with the process (Carver & Klahr, 1986), giving the students warm up practice with the technique (Ericsson & Simon, 1993; Jeffries, 1982), and scripting the prompts, I could conduct an effective think-aloud study. As well, I felt that students at this age would be less able to recall their debugging processes accurately after the fact, versus in the moment with prompts. The trade off of potentially disrupting them or adding additional cognitive load during the sessions was deemed acceptable.

For participants, I chose upper elementary students (Grades 5-7, ages 10-12) because that is the group that I interact with the most, have the most experience with, and are at a stage where they are being introduced to computer programming for the first time, i.e., they are novices in both learning to program and their general cognitive development. As well, this group is significantly underrepresented in the debugging literature, and given the push to introduce computer programming to students of this age, this seems like a timely and necessary gap to fill. I also chose students who had already taken at least one Python programming course because they would know how to complete basic program creation and would be comfortable using the tools (to some extent). Also, because of my previous and continuing work with students in this age range, they were easier to source as participants in the study.

For materials, given that there are limited studies looking at debugging with children, I modelled my course and study design after Carver's 12-week course discussed in her thesis (Carver, 1986) but updated it to use a more modern language and programming environment. I continued the use of turtle graphics as the topic to be learned during my 8-week course, because it is a topic of interest to the target audience and there are a number of existing debugging examples that I used as inspiration for my three debugging challenges used in the study.

Ultimately, I chose to look at debugging instruction for children because I have observed that it is a common place of confusion for them, and thus that it acts implicitly as a barrier to entry to programming. As a result, I wanted to develop and test the effects of providing them with explicit instruction with the goal of empowering them to take control over the process and help them develop confidence programming, or more simply, I wanted to teach them how to navigate there way across the lava.

Chapter 3.

Python Turtle Graphics Course Design

This chapter describes the design and pedagogical decisions that went into the Python turtle graphics course that provided the opportunities for data collection required for this study. This chapter is presented from the perspective of a teacher-researcher, and thus focuses on the pedagogical choices for the instruction as they relate to the research questions. For example, the decision was made to run this study as part of a course so that students could learn the debugging processes and be tested on it in a supportive context, rather than simply presenting participants with the debugging challenges and gauging their success in solving them without any instruction or opportunity to learn.

To start, turtle graphics is a series of commands that a programmer uses to draw images on a computer screen. For example, Figure 3.1 demonstrates five sample images of increasing complexity that were generated by the students during this course. Turtle graphics originated with the Logo programming language in the late 1960's (Feurzeig et al., 1969) and over the years the turtle graphic commands have lived on by being added to newer programming languages such as Python. I chose to use turtle graphics for this study partially as an homage to early debugging research, which often used Logo and turtle graphics, but also because it provides a simple set of commands that students can experiment with to create their own images.

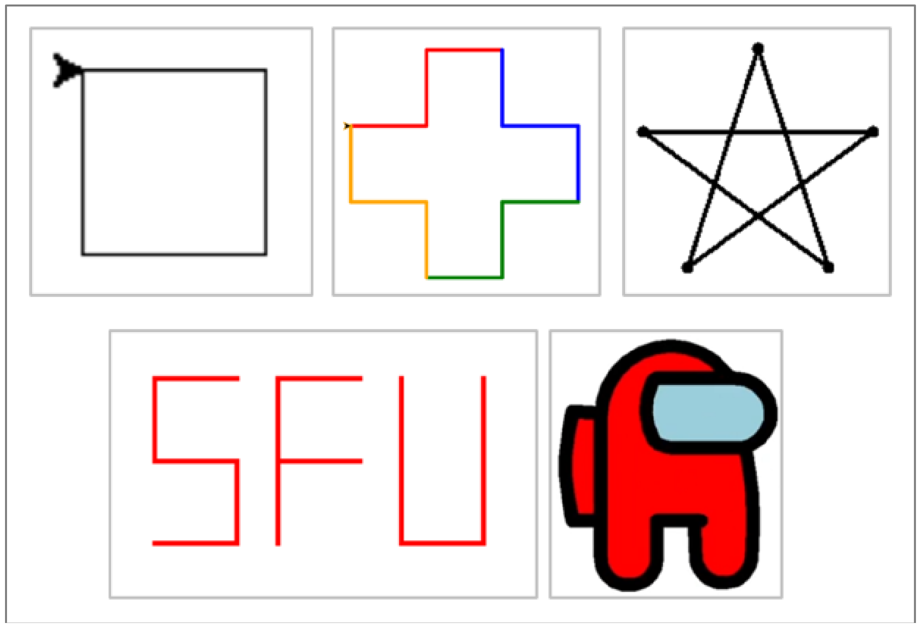


Figure 3.1. Sample images generated during the course using turtle graphics.

Additionally, inspiration for the debugging instruction implemented in the course come from Sharon Carver's PhD thesis (Carver, 1986) and detailed in Carver & Klahr (Carver & Klahr, 1986). For example, key design choices for the session progression (Carver & Klahr's was 12 weeks, mine was 8), as well as the choice to ask students to describe and speculate (they called these "bug proposals", p. 509), which were part of the Weeks 4/5 experiment protocol, were inspired by Carver & Klahr (1986). Additionally, some of the turtle graphics demonstrations and exercises were modelled after examples in Carver's papers, including Figure 3.15, the flower (Carver, 1986, p. 99). Although the general outline of my debugging class (Table 3.1) was similar to Carver's, the content is all original, and focuses on introducing debugging throughout the study through both explicit demonstrations and lessons, as well as implicit usage of debugging processes throughout.

Table 3.1. Study weekly outline

Week	Topics	Data Collection
Week 1/1a* 2022-10-19 *2022-10-25*	- Study Intro - Syntax Debugging - Turtle Graphics Intro	Question 1a: Describe Python Coding Question 2a: Describe Debugging (pre-debugging) Student Debugging Sessions Metrics & Times Student Self-Efficacy Scores (pre/post)
Week 2 2022-10-26	- Turtle Graphics: Controlling the Turtle's Pen & Colors - Expert Debugging Demonstration (SGE)	Question 2b: Describe Debugging (after basic debugging) Student Generated Examples (SGE) Question 3: Novice vs. Expert debugging differences.
Week 3 2022-11-02	- Turtle Graphics: Variables and Loops - Debugging using print statements	*No data collected this week.*
Week 4 2022-11-09	- Turtle Graphics: Drawing Stars - Draw Cross Challenge	Student Debugging Sessions Metrics & Times Student Self-Efficacy Scores (pre/post)
Week 5 2022-11-16	- Turtle Graphics: Drawing Stars (continued)	Student Debugging Sessions Metrics & Times (continued from Week 4) Student Self-Efficacy Scores (pre/post)
Week 6 2022-11-23	- Turtle Graphics: None - Explicit Debugging Process/Steps	Question 1b: Describe Python Coding Question 2c: Describe Debugging (after 2x debugging challenges)
Week 7 2022-11-30	- Turtle Graphics: Flower Shape Filling & Functions	Student Debugging Sessions Metrics & Times Student Self-Efficacy Scores (*not collected*)
Week 8 2022-12-07	- Turtle Graphics: AmongUs YouTube lesson	Student Debugging Sessions Metrics & Times (continued from Week 7) Student Self-Efficacy Scores (*post-only*)

* Week 1a: One student missed the Week 1 class, so they attended a make-up class (named Week 1a). The same material was taught during Week 1a as Week 1.

3.1. Classroom Programming Environment

Before describing the exercises for the individual weeks, this section will briefly describe the computer programming environment used by the students on their own computers. All students installed two software applications before the start of the class: Python 3 and PyCharm. The first was the Python 3 programming language itself which allowed them to run Python programs on their computers. The seconds was the PyCharm Community Edition programming editor (see Figure 3.2), which was used to write, edit, and run their Python programs. PyCharm was used because it provides a consistent cross-platform programming environment that the students had already used

in previous programming courses with me. Of particular importance for this study, PyCharm provided distinct areas for code editing ("A" in Figure 3.2) and errors ("B" in Figure 3.2).

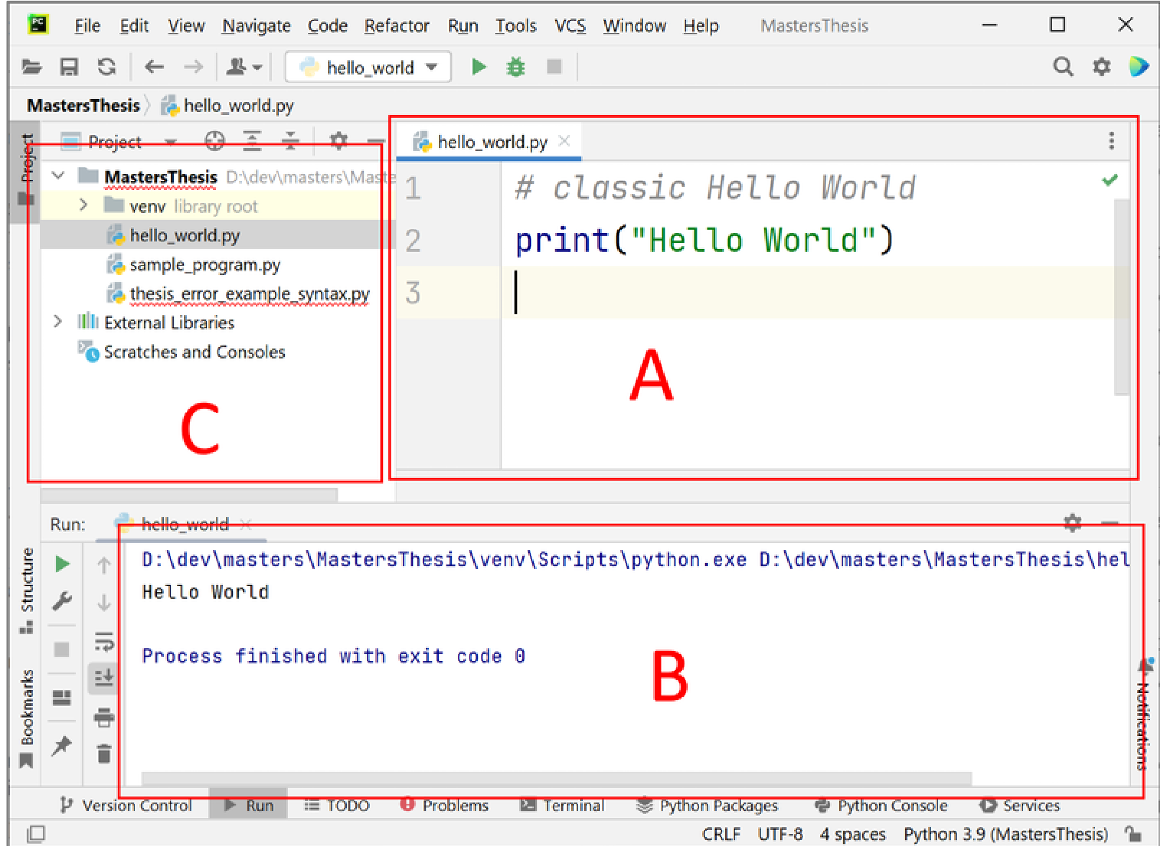


Figure 3.2. PyCharm Community Edition integrated development environment. (A) The code editing window for the script named "hello_world.py". (B) The output window for program output as well as compiler or runtime error messages. (C) The project files, including the Python script files.

In addition to the editor, this course focused on generating turtle graphics using a series of commands for drawing a turtle cursor on a computer screen. After the students had written their programs (a series of commands), they would run their program by clicking the green "run" button in PyCharm and if there were no errors, then a new window would appear on their computer screen and draw the turtle output. If, however, there were errors, then they would appear in the error area of PyCharm ("B" in Figure 3.2).

3.2. Week 1: Study Introductions and Initial Debugging Challenges

The goal for Week 1 was to introduce the students to the study, the concept of debugging, and turtle graphics. At the end of the first class, students were given an initial debugging assessment in the form of four basic challenges: three syntactical, and one semantical. From a data collection standpoint, the challenges were meant to provide an initial baseline of the student's pre-existing debugging skills and captured their initial debugging self-efficacy measures, on a scale of one to seven[14]. The challenges also provided students with practice thinking aloud while debugging.

Week 1 introduced students to the basics of setting up their turtle output window, drawing lines, and changing the turtle's heading. By the end of the first class, students had coded along with me as I live-coded how to draw two simple shapes (a square and a triangle) using turtle graphics (Figure 3.3). The live coding consisted of me sharing my screen with them, writing each line of code, and periodically running the program on my computer so they could compare their output to mine.

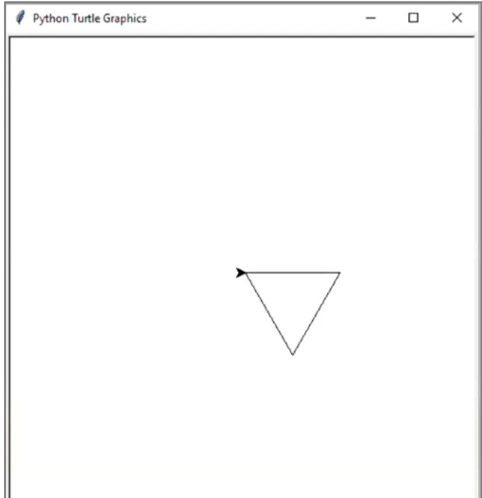


Figure 3.3. Sample turtle graphics output of a triangle.

In addition to the turtle graphics introduction, students were introduced to three types of errors that can occur when writing a computer program: syntax errors, runtime

[14] The self-efficacy scale was initially scored out of seven rather than ten such that students had to either choose a preference towards lower (one) or higher (seven), similar to a Likert scale. However, in practice, some students chose fractional values, especially around the mid-point of 3.5.

errors, and semantic errors. Syntax errors occur when the written code is misspelled or malformed in some way and result in a compiler error message. Runtime errors occur when the program attempts to do something that is not permitted while running and result in the program failing to run or crashing and returning a runtime error message. Semantic errors occur when the program's output does not match the intended or required target output but unlike the other two errors, the program runs but does not produce the desired results. Students were introduced to each error type by way of an example that they coded along to with the teacher. As a pedagogical device, the teacher would write valid code, ensure that it was correct and then instruct students to explicitly introduce the desired bug and run the program to see how it manifest.

For example, as a demonstration of a syntax error, instead of the correct line: *print("hello world")*, as shown in Figure 3.4, they were told to change the letter "i" in print to another vowel, like "o" which results in the incorrect function name "pront". They were then told to notice how the editor (PyCharm) underlined the now incorrect function name with a red underline, as shown in Figure 3.5, as a means of notifying them of the error. Next, they were told to run the program, which would produce a NameError compiler error message as shown in Figure 3.6. They were told to read the compiler error from the bottom to the top, starting with the NameError line at the bottom, looking at the identified code, if any, and then noting the line number where the error was identified. The goal with this demonstration was to provide the students with a worked example (Merrill, 2002; van Merriënboer et al., 2002) that they themselves had control over such that they could see what had caused the resulting error. This removed the need to locate the error because they already knew where it was located. Finally, students were told to repair the error and run the program again to ensure that their repair was correct and that the program was working as expected again.

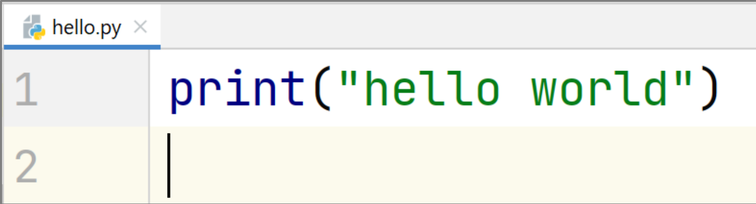


Figure 3.4. Sample Python hello world program code.

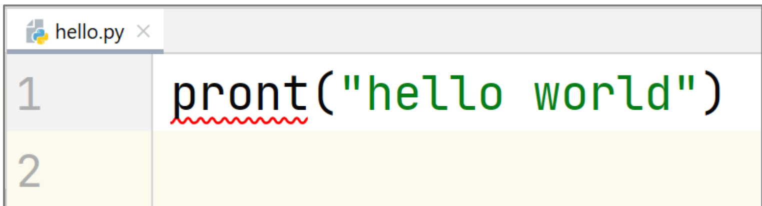


Figure 3.5. Sample of bugged NameError version of print statement.
The correct function name “print” has been intentionally replaced with the incorrect function name “pront” to explicitly generate the red underlining of the error by the PyCharm editor.

```
Traceback (most recent call last):
  File "hello.py", line 1, in <module>
    pront("hello world")
NameError: name 'pront' is not defined
```

Figure 3.6. Sample NameError compiler error message (Week 1).

At the end of the first hour, students were randomly selected to individually move over to a Zoom breakout room to participate in a series of video-recorded one-on-one debugging challenges. The remaining students were asked to wait on the Zoom call until their turn arrived and dismissed once they had completed their session. Each one-on-one session started with the student being asked to rate their level of confidence debugging Python code, from one to seven (with one being the least confident). Next, students were given a URL via Zoom chat to click on that led them to the four debugging challenges. (The code for the four challenges can be found in Appendix B.) These were novel challenges that they had never seen, which had been produced by me. The student was instructed to copy the code for the first challenge, create a new Python script titled “week1_challenge.py” and paste the challenge code into the new script and run it. They were asked to run the first challenge to ensure that they had switched their PyCharm to run this new challenge script each time rather than the class script that they had previously been working on. During each of the four challenges, student was asked to speak aloud while debugging to tell the teacher-research, “what they were seeing, thinking, and doing.”

The debugging challenges for Week 1 did not use turtle graphics, but instead focused on syntax and semantic errors using print-statements and variables. This choice was made in an effort to reduce the amount of code each student needed to read and debug and to focus on what strategies students used for syntax and semantic errors at the start of the class. It should be noted that most studies on “novice” programmers avoid or ignore syntax errors, because they are considered too basic for undergraduates or above participants. I felt that it was necessary to observe the participants debugging syntax errors because Carver’s similarly-aged students and teachers (Carver, 1986; Carver & Risinger, 1987) struggled with them during her studies. It should be noted that the PyCharm editor used for this study provides a great deal of feedback to the programmer, including error underlining (see Figure 3.5), which is a common feature of code editors today but was not available during Carver’s studies.

3.3. Week 2: Pens, Colors, and Student Generated Examples

The goal for Week 2 was to introduce the students to turtle graphic functions that allowed them to change the size and colour of the pens used to generate their line drawings. As well, students were tasked with generating buggy versions of their programs, sending those buggy versions to the teacher via the online chat feature of Zoom, and then watching the teacher attempt to debug them.

The turtle graphics goal for Week 2 was to introduce students to colours in turtle graphics, starting with the colour of their pens, and then additional related functions like pen size, raising and lowering the pen (using the penup and pendown commands), and more (see Figure 3.7). By the end of Week 2’s instructional hour, students were creating their own colourful line drawings and were encouraged to experiment and create their own drawings.

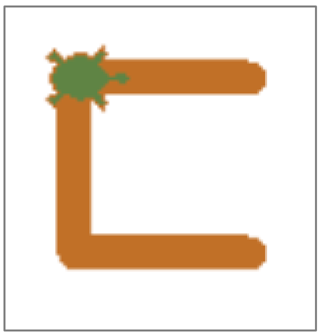


Figure 3.7. Week 2 sample brown coloured shape with green turtle.

Before introducing students to any explicit debugging instruction, I chose to first demonstrate my own debugging approach using code they created. To start, the students were instructed to create or modify their Week 2 program to create a buggy version with no more than three errors of their choice and then to send their bugged versions to me via Zoom chat. I then randomly chose a student, asked them to share their screen with the class, and run their bug-free version of their program to demonstrate what the output should look like. I took a screenshot of the bug-free version's output, shared my screen with the class and then proceeded to attempt to debug the buggy version with only the buggy source code and a picture of the target output. This demonstration was offered to the students as both a challenge to see if they could stump the teacher as well as an attempt to demonstrate an expert debugging process when presented with code written by others.

I modelled my own think-aloud process as I reviewed the error message, output, or code (as appropriate) to identify, locate, and repair the bugs. Each debugging attempt started with me running the program to see what compiler or runtime errors were reported or to see what output resulted. Then, a hypothesis or target area of the code was identified based on the error message or output observed. After each repair attempt, I would re-run the program to reassess my repair and progress towards matching to the target output. Finally, when I felt that I had successfully debugged the student's program, I would ask the student if my result was correct.

In all but two of the student programs, I successfully debugged the programs using this approach. Of the two where I did not initially debug the code successfully, one involved the student sending the wrong code and the other involved the student editing a hexadecimal colour code that did not match the one I chose exactly. In the first

instance, I did successfully debug the code the student provided and discovered a bug in the program that the student had unintentionally introduced. In the second situation, the image produced when the student demonstrated their correct version resulted in an image that did not reproduce the colour the student had used. This may have been the result of the Zoom screensharing resulting in an approximation of the student's chosen color and thus when I sampled the colour using an image tool, I obtained a colour that visually was similar, but which did not match the student's chosen colour code exactly. This led to a discussion of the importance of target specifications being exact and matching to that target.

At the end of my demonstration, students were asked to provide a description of the difference between the way they debugged code written by others and the way that I had debugged code written by others. The results of this question are provided in the Results chapter.

3.4. Week 3: For-Loops and Variables

The goal for Week 3 was to introduce students to for-loops in Python so they could create increasingly complex drawings using loops rather than repeatedly writing out the same code. In addition, a second debugging challenge was scheduled that included a looping bug. Unfortunately, a number of students were late for the class, three by ten to twenty-five minutes and one by an hour. As a result, the originally scheduled Week 3 debugging challenges were moved to Weeks 4 and 5.

As part of the introduction to loops this week, we started with a new script that did not use turtle graphics, so we could focus on the loop mechanism itself. To do this, we created a new for-loop with a range of ten, and within the loop we printed the value for the loop variable (see Figure 3.8). Before we ran the program, students were asked to guess what would be printed, and specifically what the starting and ending values would be. Many of the students were unaware that the loop would start with zero and end with the value specified in the range function minus one, i.e., nine when the range value of ten was specified. This initial explicit program called loops.py was further augmented to demonstrate various features of the range function so that students both experienced how for-loops in Python work and how to modify them to achieve different

results. Students were also being primed for the subsequent challenge, which contained an example of a for-loop off-by-one bug.

```
1  # loops.py
2  for i in range(10):
3      print(i)
```

Figure 3.8. Week 3 loops.py – print the loop values.

After reviewing for-loops without turtle graphics, we switched back to turtle graphics for Week 3 to revisit the drawing of polygonal shapes using loops and variables. As shown in Figure 3.9 below, we first reviewed and then isolated the repeated code previously written to draw a square: forward(100) and then right(90) four times. We then created a for-loop that looped four times to achieve the same square as before. Next, we replaced the line length (100 in the example) with a variable so that we could change the size by changing the value of the *size* variable.

```
5   # repeated code x4
6   forward(100)
7   right(90)
8   forward(100)
9   right(90)
10  forward(100)
11  right(90)
12  forward(100)
13  right(90)
```

```
5   # loop x4
6   for i in range(4):
7       forward(100)
8       right(90)
9
10
11
12
13
```

```
5   # loop x4 w/ size var.
6   size = 100
7   for i in range(4):
8       forward(size)
9       right(90)
10
11
12
13
```

Figure 3.9. Week 3 for-loop progression (Python).

Although not shown, we continued to create a variable named angle and set it to the value of 90 (angle = 90), and then used this new angle variable to arrive at the same result once again. Then, we created one more variable named *sides*, and set it to four (sides = 4) and replaced the value within the for-loop's range with the new sides variable: *for i in range(sides)* and again ran the program so we again drew the same square. Finally, after all of these modifications, we changed the values of sides to five (sides = 5) and angle to 72 (see Figure 3.10). When we ran our program, we produced a five-sided octagon instead of a square. Students were then tasked to explore changing the variables to see what other shapes they could create.

```
5   # final week 3 for-loop octagon
6   size = 100
7   sides = 5
8   angle = 72
9   for i in range(sides):
10      forward(size)
11      right(angle)
```

Figure 3.10. Week 3 Final octagon Python code with variables and a for-loop.

For the Week 3 debugging challenge (Figure 3.11), I chose to not include any syntactical errors. Instead, I chose to focus on asking students to predict what they thought might be the bugs that caused the differences they noticed between the target output and the actual output *before* they could see the code. I wanted to focus on their debugging processes rather than their apparent ability to spot the differences.

For this challenge, I introduced four bugs, one of which was repeated twice (see Appendix B for actual code used). The first and third bugs were duplicates of each other and involved using a hardcoded constant value of one in place of a variable to specify the size of the pen used to draw the lines for the squares. The goal was to see if students would identify the duplicate error and apply the same fix twice or not, and if they would notice that the *pen_size* variable had been used for the first square and thus logically should be used for the other squares too, as noted in the challenge instructions. The second bug was to add a negative sign in front of a variable that would cause it to invert on the x/y axis, as demonstrated by the green square. The goal here was to observe how students would locate the bug and then how they would identify the cause and correct it. It should be noted that all but one student identified the issue and corrected it by removing the minus sign. The one student who did not identify the minus sign instead inverted the turtle using a *right(180)* command before drawing the second and third squares, and thus successfully recreated the target image but did so in a different manner. Finally, the fourth bug involved a for-loop off-by-one bug where the range value for the blue square (square three) only counted to three: *range(1, 4)*, rather than four like the other lines: *range(4)*. The goal here was to see whether students identified the issue with the range and what repair they would apply to the line.

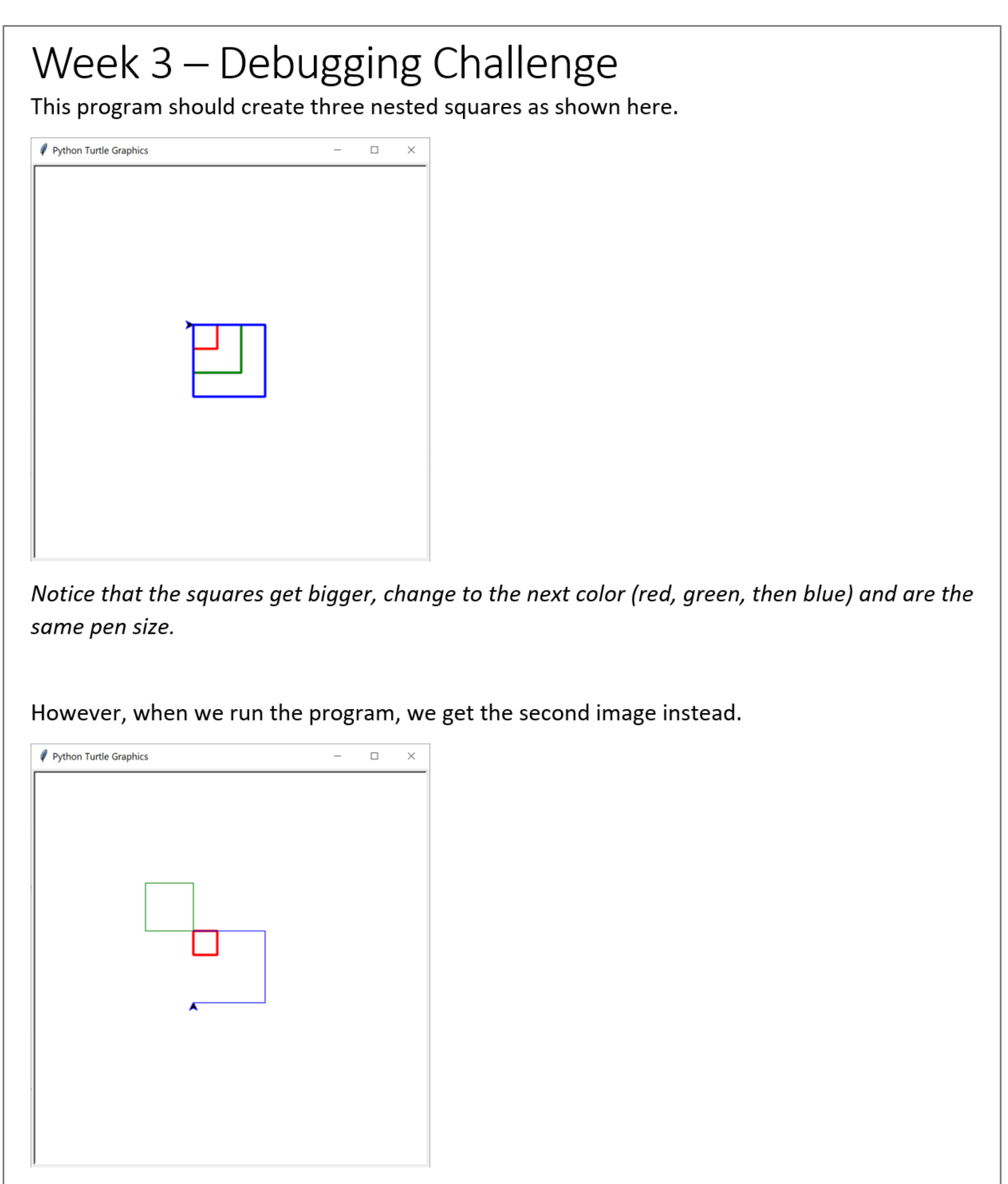


Figure 3.11. Week 3 debugging challenge.

3.5. Week 4: Shapes: Stars, Crosses, and Challenges

The goal for Week 4 was to continue to use for-loops to create shapes with repeating patterns. Students used variables for the colours, the lengths, and the angles and then combined those within a for-loop to create a star. Once they had their five-

point stars working, they were tasked to modify them to create a cross shape while they waited their turn for the debugging challenge (originally scheduled for Week 3).

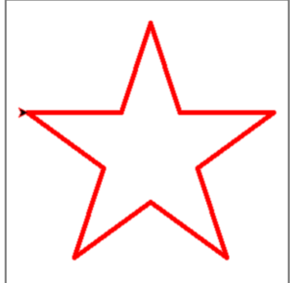


Figure 3.12. Week 4 five-point outlined star shape using for-loops.

In addition to continuing to use for-loops with turtle graphics, students were also introduced to the idea of exposing the values of variables as they changed, using print statements, something I call “debug print-statements” because their role is to expose the state of the machine to inform debugging. Thus, during the creation of their five-point stars, two additional variables were added: *total_length* and *total_angle* and their values were printed out to the console while the stars were being drawn. The two total values were chosen to show the students how they could see the changes in their variables while looping, and also mark areas of their code with debug print-statements that they could use to identify which part of their code was involved and in what order it executed. These two features (identifying the order of operations and the changing value of variables) were repeated but were not called out as necessary during their debugging challenges. However, within the debugging challenge itself, debug print-statements were included in the form of statements that printed before a shape was drawn, such that students could, if they wished, add code either before or after these markers so they could determine if a bug happened before or after a certain point.

3.6. Week 5: Turtle Stars, Functions, and Challenges

The goal for Week 5 was to introduce students to Python functions so they could draw multiple instances of the same shape simply by calling a named function rather than duplicating the necessary code. Once again, debug print-statements were the focus of the explicit debugging instruction. Additionally, although not original scheduled, students who did not complete the debugging challenge during Week 4, due to time, completed it this week.

Students started Week 5 by making a copy of their star code from the previous week into a new file so we could convert it to a function. We then modified the code to create a function named "star" and placed their previous star-drawing code inside the new function. Next, the line length information was changed to a function parameter and the function was called multiple times to create multiple stars, as shown in Figure 3.13 below.

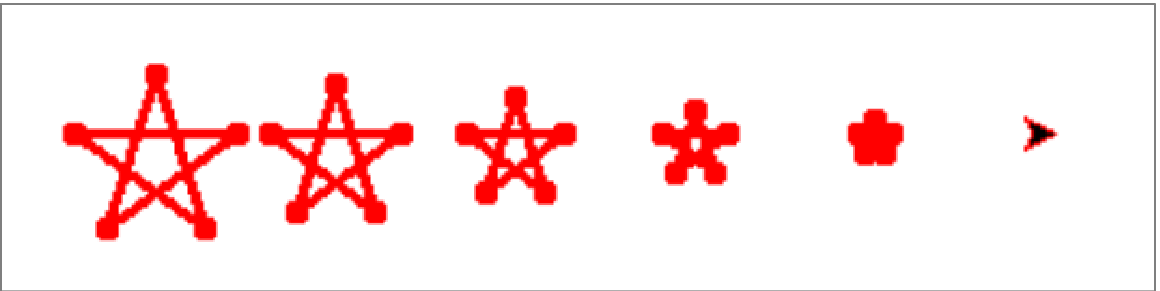


Figure 3.13. Multiple five-point connected-point stars created using a stars function.

Again, similar to Week 4, debug print-statements were added to the function to print each time it ran as a way for students to observe that the function was being called multiple times. This instruction was meant to prime students for the following week's use of functions with print-statements.

3.7. Week 6: Bug Localization with the Wolf Fence Algorithm

The goal for Week 6 was to introduce and practice explicit debugging processes for localizing bugs in code using debug print-statements.

The first process reviewed during Week 6 involved segmenting the code in order to divide-and-conquer the search space effectively, a technique known as the "Wolf Fence" algorithm (Gauss, 1982). Students were first introduced to the algorithm as a set of steps for progressively dividing the code in half such that one half has been determined not to contain the bug (the wolf) which means it must be in the other half (the other side of the fence). After introducing the algorithm, students were given a buggy code sample and then shown how to insert debug print-statements so that they could divide the code up to locate the bug.

After trying the Wolf Fence algorithm for locating bugs, students were tasked to try to debug some provided code by first generating a hypothesis before trying to debug it. In this exercise, once they had taken a guess, they were given a chance to search for the bug using the techniques and tools provided, and then asked how their hypothesis matched up. After trying the hypothesis approach, students were given code that was supposed to draw four squares but instead produced three squares and a triangle, as shown in Figure 3.14. We then engaged in a group debugging challenge where I attempted to use debug print-statements and the Wolf Fence algorithm to debug the code. We ended the class after completing this exercise because two students had to leave early for other engagements.

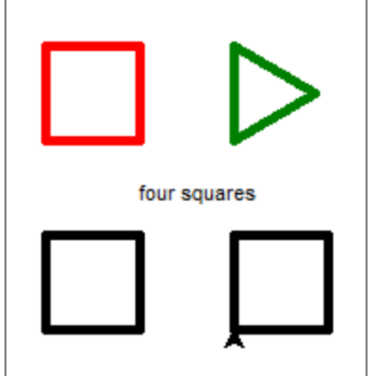


Figure 3.14. Week 6 "four squares" debugging demonstration.

3.8. Week 7: Complex Shapes with Functions

The goal of Week 7 was to explore the creation of a moderately complex shape using all of the techniques already covered during the class. The output was inspired by a similar flower created by Carver (1986) using turtle graphics and Logo (Figure 3.15). The students followed along through the worked example but modified their flowers based on their own preferences and ideas.

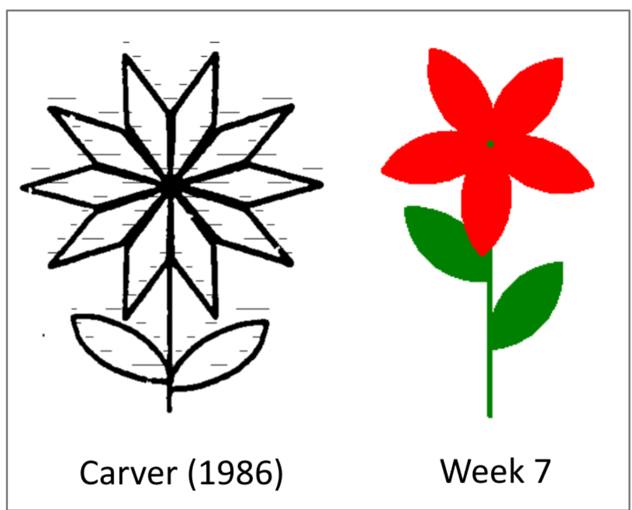


Figure 3.15. Week 7 turtle graphics flower example.
The “Carver (1986)” flower is from Carver (1986, p. 99).

I had originally planned to repeat the stump the teacher challenge from week two, to see what different types of bugs students would introduce. However, we ran out of classroom time building the flower and instead I used the last forty-five minutes to start the final debugging challenge.

For the final debugging challenge, I decided to focus on introducing two logic bugs that would require the students to trace code using the explicit debugging techniques covered during the course. To do this, I used two Python programming techniques not covered in the class, lists and the modulo arithmetic[15] operator (%). The list was used to store a list of colours, and the modulo operator was used to determine if a number was even or odd.

Knowing that the students were most likely unfamiliar with these two techniques, I provided pre-built debug print-statements such that students could uncomment any of the four provided debug print-statements to discover what these new statements did and use that information to locate and correct the two bugs. Code comments were used throughout the course and students were told they could uncomment any of the code they felt they needed. In addition, during the one-on-one debugging challenges, I explicitly reminded each student that the debug print-statements were present, and I answered any questions they had about modulo arithmetic or the list of colours. The

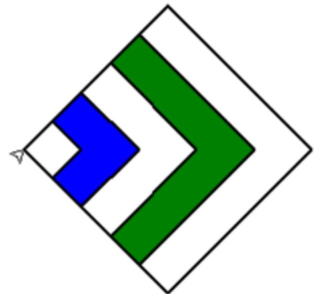
[15] The modulo arithmetic operator in Python is the percentage sign “%”. The logic used was “if number % 2 == 0” then the number is even, else it is odd.

challenge was for them to try to locate the bugs using the explicit debugging techniques covered in the course.

Week 7 – Logo Bug Challenge

You've been asked to draw a logo using turtle graphics. The logo should look like this:

However, the code you have is showing this instead:



The code used to work, but some things were changed and now it's broken. Try adding some debugging code using the print function to help figure out why the logo code is broken so you can fix it.

Figure 3.16. Weeks 7 & 8 final debugging challenge, the "Logo Bug Challenge".

3.9. Week 8: Complex Shape: Among Us Character (Video)

During the first class (Week 1), students were shown a slide with a variety of drawings that had been generated using turtle graphics. One of the drawings was for a character from the video game *Among Us* which the students called, "the Among Us guy" (see Figure 3.17). In place of a live-coding session during Week 8, I created a

YouTube video[16] of the lesson and made that available to the students to follow along to while I conducted the final one-one-one debugging sessions. I used a video lesson because I was trying to squeeze in the final debugging challenge with the remaining seven students. I had expected students to take an average of fifteen minutes each to complete the challenge, but the final two students combined took approximately forty-five minutes instead. Because the final two students were using the debugging techniques taught in the course, I wanted to gather as much data as I could, and thus decided to drop the final three students.

No new explicit debugging instructions were presented during Week 8; however, debug print-statements were used throughout the *Among Us* video lesson.

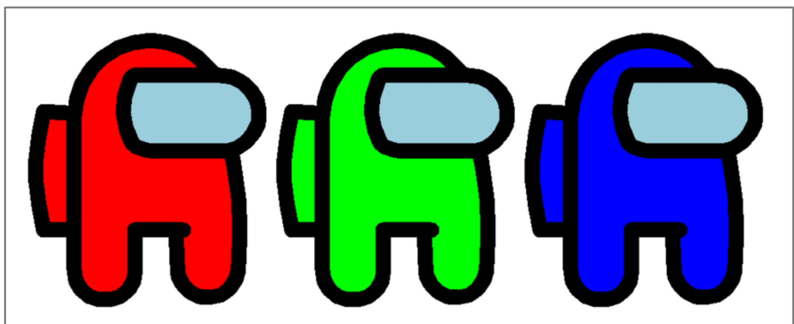


Figure 3.17. Week 8 three Among Us character drawn using turtle graphics.

[16] The author's "Drawing Among Us characters with Python Turtle Graphics" video https://www.youtube.com/watch?v=W7XZfdT4V9g

Chapter 4.

Methodology

This chapter describes the research approach and methods used to investigate and describe how a group of elementary students debugged Python programs during an eight-week programming course. The chapter proceeds as follows. The first section describes the research questions, framework, setting, and recruitment. The second section describes the debugging environment and programming bugs for each challenge. The third section describes the questions that students were asked at various stages throughout the study. The final section describes the data collection and data analysis approaches.

4.1. Research Questions

This study aims to identify what debugging behaviours and approaches novice programmers exhibit prior to debugging instruction and what impact teaching them explicit debugging techniques has on their subsequent debugging activities. This focus has led to the following four research questions:

RQ1: What strategies do novice elementary school students employ when debugging Python code?

RQ2: What challenges do novice students face when debugging?

RQ3: What effects does an explicit debugging strategy have on student debugging strategies?

RQ4: In what ways do students respond to explicit debugging instruction?

4.2. Research Framework

This research aims to identify what strategies novices come to debugging with before explicit debugging instruction and then evaluate the impact of teaching them explicit debugging techniques. As a result of these two goals, I decided to follow a

qualitative research approach so that I could capture and describe the observed details and then compare them throughout the study.

In line with previous debugging studies, for example Vessey (1984), Nanja (1988), and more recently Whalley et al.(2023), I chose to analyze verbal protocol data (Ericsson & Simon, 1993). To do that, I chose to gather not only participants' observational data recorded via their computer screens, but also their concurrent verbalized statements about their thought processes. This approach is often called concurrent think-aloud and is done while the participants are performing the task. The goal with this form of think-aloud is to capture the participants' immediate thoughts and cognitive processes during problem solving rather than just the outcome from problem-solving.

Following the collection of the think-aloud data above, I started with a grounded theory-based approach (Charmaz, 2006) similar to Fitzgerald et al. (2005) as a starting point for the data analysis. However, unlike Fitzgerald et al, rather than attempt to develop a theory based solely on my own interpretation of the collected data, I leveraged an existing codebook from Vessey (1984) to arrive at a combination of inductively and deductively discovered strategies and challenges. This blended approach resulted in improved consistency between coders and as a result a more complete set of debugging strategies and challenges.

4.3. Research Setting

To better understand the strategies and challenges that novice programmers face when debugging, it was crucial to directly observe them while they attempted to debug computer programs. The research questions were pursued by providing students with buggy computer programs and observing their actions and their descriptions while they attempt to debug them.

SFU's Office of Research Ethics (ORE), gave approval to conduct the research on September 23, 2022. The research was conducted online via the Zoom video-conferencing platform over an eight-week during the Fall of 2022, from October 19, 2022 to December 7, 2022. The participants were all elementary students, ages 10-12, from two urban school districts in British Columbia, Canada.

4.4. Recruitment

To recruit participants, I employed non-random convenience sampling by including a notice about my research study in an email to parents of students who had previously taken after-school classes with me before. In early October 2022, I sent out approximately one hundred emails advertising my Fall 2022 after-school educational technology programs. This email announced my Minecraft and Python programming classes but also included in the body of the email a statement announcing my research study, see Figure 4.1.

My SFU research project:

This fall I will be conducting an online research study on Wednesdays after school as part of my master's in education at SFU. This study involves a free 8-week online Python programming course and is open to students in Grades 5 to 7. To learn more about the study, email me or visit: https://www.xmodus.com/sfu-study-fall-2022.html

Figure 4.1. Email recruitment statement.

The text for the recruitment web page noted in Figure 4.1 can be found in Appendix E. Sixteen parents expressed interest and from those, nine ultimately signed up. Parents gave their consent to participate (see Appendix F), and children agreed to participate after giving their assent (see Appendix G). In contrast to my after-school programs, of whom these parents and students were previously associated with, no fee was charged for participation in this class and participants were not provided with any incentives to participate besides free participation in the class.

4.5. Debugging Environment

The eight-week class in which the study took place met once a week on Wednesdays via Zoom from 3:30 PM to 5:30 PM. Each student ran their own local copy of the PyCharm Integrated Development Environment (IDE)[17] on their own computer and had the Python 3 programming language installed with it.

[17] https://www.jetbrains.com/pycharm/

The programs that the student had to debug during weeks 1, 4/5, and 7/8 (described below under Debugging Tasks) were created by the researcher and not shown to the students prior to their debugging sessions. Instead, during each one-on-one debugging session. students were given a URL via the Zoom chat feature to access the bugged code as a text file on a web server (see Appendix B for the code samples), which they then copy-and-pasted from the web page into their PyCharm editor. For Weeks 4/5 and 7/8, the students were also provided with a description of the target turtle output (see Appendix B) in the form of a PDF document, which was not necessary for Week 1 because the code did not include any turtle output.

During the one-on-one debugging sessions, participants were instructed to share their local computer screen via Zoom, and to speak aloud as they debugged their programs within their PyCharm editors. All of their actions and utterances were recorded within Zoom and downloaded later for analysis.

4.6. Debugging Challenge Bugs

During the course, only six of the nine students[18] participated in three one-on-one debugging challenges which consisted of buggy code and a description of what the code was supposed to do. Students were tasked to share their screen via Zoom and speak aloud while they attempted to debug the program. Participants were not told how many or what type of errors the programs contained.

Since the aim of the research was to determine the debugging approaches of the participants, the programs chosen to be debugged were similar to programs the students had already written during the course described previously in Chapter 3. This was intentional, so as to not unintentionally introduce code that was unfamiliar to the students. This section will describe the bugs for each challenge. All of the programs used are presented in Appendix B, along with a description of each bug and the fully corrected version of each program.

[18] During Weeks 7/8, three students did not participate in the challenge due to time constraints.

4.6.1. Week 1: Debugging Challenge #1

The Week 1 challenges were made intentionally easy so that students could get used to speaking aloud while debugging. As suggested by Ericsson & Simon (1993), the students were given time to become used to speaking aloud while debugging by practicing the process during Week 1. Participants were not told how many or what type of bugs each of the challenge programs contained. Participants attempted to debug four challenges, as shown below.

The first debugging challenge during Week 1 involved two syntax errors, lines 3 and 4 (Figure 4.2). If the student ran the program, they would first receive an error message on line 3 with the error text: *NameError, name 'Age' is not defined*. Once line 3 was corrected, if they ran the program again, they would receive an error message on line 4, with the error text: *NameError: name 'a' is not defined*. Students had to identify that the variable "Age" was incorrectly capitalized on line 3 and that the variable "a" on line 4 should be the correct variable "age" instead. The corrected version of the first challenge is shown in Figure 4.3.

```
1  # Challenge #1
2  age = 1
3  age = Age + 1
4  print(a)
```

Figure 4.2. Week 1 debugging challenge #1 buggy code.

```
1  # Challenge #1
2  age = 1
3  age = age + 1
4  print(age)
```

Figure 4.3. Week 1 debugging challenge #1 corrected code.

4.6.2. Week 1: Debugging Challenge #2

The second debugging challenge during Week 1 involved a single syntax error, where students had to identify that the function *print* was incorrectly capitalized as *PRINT on line 3*. If the student ran the program, they would receive an error message on line 3 with the error text: *NameError: name 'PRINT' is not defined*. Students had to change the incorrect function name *PRINT* to the correct lowercase *print* on line 3. The buggy version is shown in Figure 4.4 and the corrected version in Figure 4.5.

```
1  # Challenge #2
2  name = "Billy"
3  PRINT(name)
```

Figure 4.4. Week 1 debugging challenge #2 buggy code.

```
1  # Challenge #2
2  name = "Billy"
3  print(name)
```

Figure 4.5. Week 1 debugging challenge #2 corrected code.

4.6.3. Week 1: Debugging Challenge #3

The third debugging challenge during Week 1 involved a single syntax error, where students had to identify that line 4 was incorrectly indented. If the student ran the program, they would receive an error message on line 4 with the error text: *IndentationError: unexpected indent*. The students had to remove the leading spaces in front of the *print(after_birthday)* on line 4. The buggy version is shown in Figure 4.6 and the corrected version in Figure 4.7.

```
1  # Challenge #3
2  age = 6
3  after_birthday = age + 1
4    print(after_birthday)
```

Figure 4.6. Week 1 debugging challenge #3 buggy code.

```
1  # Challenge #3
2  age = 6
3  after_birthday = age + 1
4  print(after_birthday)
```

Figure 4.7. Week 1 debugging challenge #3 corrected code.

4.6.4. Week 1: Debugging Challenge #4

The fourth debugging challenge during Week 1 involved a single semantic error, where students had to identify that line 4 did not change the value for the variable height and so the program output the wrong final value. If the student ran the program, the program ran and output the incorrect final value of 4 instead of 6. The students had to change the code so that the final value for the variable height was 6, as printed out by line 6. The buggy version is shown in Figure 4.8 and two possible corrected versions are shown in Figure 4.9 and Figure 4.10.

1	# Challenge #4
2	height = 2
3	height = height + 2
4	**height + 2**
5	# at the end, the height should be 6
6	print(height)

Figure 4.8. Week 1 debugging challenge #4 buggy code.

1	# Challenge #4
2	height = 2
3	height = height + 2
4	**height = height + 2**
5	# at the end, the height should be 6
6	print(height)

Figure 4.9. Week 1 debugging challenge #4 corrected code (option 1).

1	# Challenge #4
2	height = 2
3	height = height + 2
4	**height += 2**
5	# at the end, the height should be 6
6	print(height)

Figure 4.10. Week 1 debugging challenge #3 corrected code (option 2).

4.6.5. Week 2: Student-Generated Examples (SGE)

During Week 2, students were given time to create their own custom turtle graphic images using the commands covered during the class. After they had all created their own custom images, they were asked to create a copy of this non-buggy image and introduce up to three bugs to create a buggy version of their own program. Then, when called upon, they ran the non-bugged version on their computer and showed the output to the class via the Zoom screensharing feature. Then, after showing the output of their non-buggy version, they provided the bugged version of the code to the teacher for debugging. This was framed as a challenge for the teacher, i.e., stump the teacher. Students were told to include a maximum of three bugs due to the time constraints of debugging nine student projects. The researcher then debugged all of the programs thinking aloud while talking through each step of their debugging process. At the end of the session, participants were asked to comment on what differences they noted between the teacher's debugging process and their own. Their descriptions are detailed in the Results chapter.

4.6.6. Weeks 4/5: Debugging Challenge

During Weeks 4 & 5, the students were first presented with a two-page PDF document labelled Week 3 Challenge (see Figure 3.11). The first page of the challenge showed both the desired target output and a description of the elements of note, as well as the actual output from the bugged version of the program. The students were instructed to read the first page and then asked to comment on what they noticed (identify), and what they thought might be the causes (hypothesize). Following that, they were instructed to download the code and begin debugging. They were reminded as they started to debug that they were to think-aloud as they debugged.

The buggy code for Weeks 4/5 contained four semantic bugs, as highlighted in Figure 4.11. The two *pensize(1)* errors should have instead used the *pen_size* variable, resulting in *pensize(pen_size)*, to produce the same outline size for all three squares. The negative sign in front of the variable *square_size* for Square #2 results in the second square rotating 180 degrees until it was removed. The final error was that the range value for Square #3 only counted to three, resulting in three sides. The final error could be corrected either by changing the incorrect range value to *range(4)* or *range(1, 5)*.

The full code listing for Weeks 4/5, buggy and corrected, are in Appendix B.

<table>
<tr><th>Square #1</th><th>Square #2</th><th>Square #3</th></tr>
<tr>
<td><pre>
print("square 1")
pencolor("red")
pensize(pen_size)
for i in range(4):
    x = square_size
    forward(x)
    print(...)
    right(90)</pre></td>
<td><pre>square_size += 30
print("square 2")
pencolor("green")
pensize(1)
for j in range(4):
    x = -square_size
    forward(x)
    print(...)
    right(90)</pre></td>
<td><pre>square_size += 30
print("square 3")
pencolor("blue")
pensize(1)
for k in range(1, 4):
    x = square_size
    forward(x)
    print(...)
    right(90)</pre></td>
</tr>
</table>

Figure 4.11. Weeks 4/5 debugging challenge bugs side-by-side.

4.6.7. Weeks 7/8: Debugging Challenge

The buggy code for Weeks 7/8 contained two different errors. The first was that the function *is_even* returned True for odd numbers rather than for even numbers, which resulted in the colours being swapped. The second error was that the pen was the

wrong colour when the squares were drawn, resulting in an incorrect outline colour. The program ran with no errors but produced the wrong out, as shown in Figure 3.16.

To fix the first error, students needed to identify that the *is_even* function was returning the wrong value and either correct the if-conditional statement, swap the return values, or change what happened when the *is_even* function was called by the main program. To fix the second error, students could have either added a command to change the pen colour to match the square colour, or they could have uncommented line 44 which change the pen colour for them.

For the Weeks 7/8 debugging challenge, the code included a set of debug print-statements (see Figure 4.12) within the main program loop that were commented out but ready for the students to use, if they wished. These disabled statements were added to see if students would take advantage of debug print-statements to see how unfamiliar code ran or to trace the value of the buggy *is_even* function. Students were introduced to the use of debug print-statements throughout the course as a debugging tool they could use to expose relevant program state, but in this case potentially relevant examples were provided but also disabled (commented out) so that students had to actively choose to use them by uncommenting them. All participants were informed of the embedded debug print-statements during their debugging sessions.

The full code listing for Weeks 7/8, buggy and corrected, are in Appendix B.

```
37        # some debugging print statements
38        # print("s", s)
39        # print("s % 3", s, s % 3)
40        # print("is_even", s, is_even(s))
41        # print("fc", fc)
```

Figure 4.12. Weeks 7/8 commented out debug print-statements.
The hashtag (#) at the front of the line marks the line as a comment and disables the code to the right. Removing the hashtag (and leading space) in front of lines 38-41 would re-enable them.

4.7. Student Questions

In addition to the debugging challenges, students were asked a series of questions at various points during the course, which they responded to via the chat function in Zoom or verbally, depending on the question. Each of the questions are

described next, their data is described in the Findings chapter, and discussed in the Discussion chapter.

4.7.1. Student Self-Reported Self-Efficacy Scores

During Weeks 1, 4/5, and 7/8, students were asked to rate their level of confidence debugging Python programs on a scale from one to seven, with one being the lowest and seven the highest. For Weeks 1 and 4/5, students were asked at the start of the challenge (pre-) and after completing the challenge (post-). For Weeks 7/8, the students were only asked at the end of the challenge and the first two students during Week 7 were not asked due to researcher error. Students were permitted to provide fractional responses, i.e., 3.5 out of 7. In addition to the reported numeric values, students were also asked to provide a reason for why the number changed (or not) between the start of the challenge and the end. The ratings and the change-reasons were captured verbally.

4.7.2. Student Debugging Description Question

During Weeks 1, 2, and 6, students were asked to provide a one-to-two sentence description of debugging. The question was phrased as: “In a sentence or two, how would you describe debugging to a friend?” Students wrote their responses using the Zoom chat function.

4.7.3. Student Python Coding Description Question

During Weeks 1 and 6, students were asked to describe Python coding to a friend in one or two sentences. This question was originally meant as a warm-up question for students before they answered the debugging description question mentioned above. The data from this question was ultimately not used in this study.

4.7.4. Student Description of Debugging Difference with Expert

During Week 2, after watching the teacher debug their buggy programs, students were asked to describe the difference between the expert’s approach and their own. They provided these one or two sentences via the Zoom text chat.

4.8. Data Collection

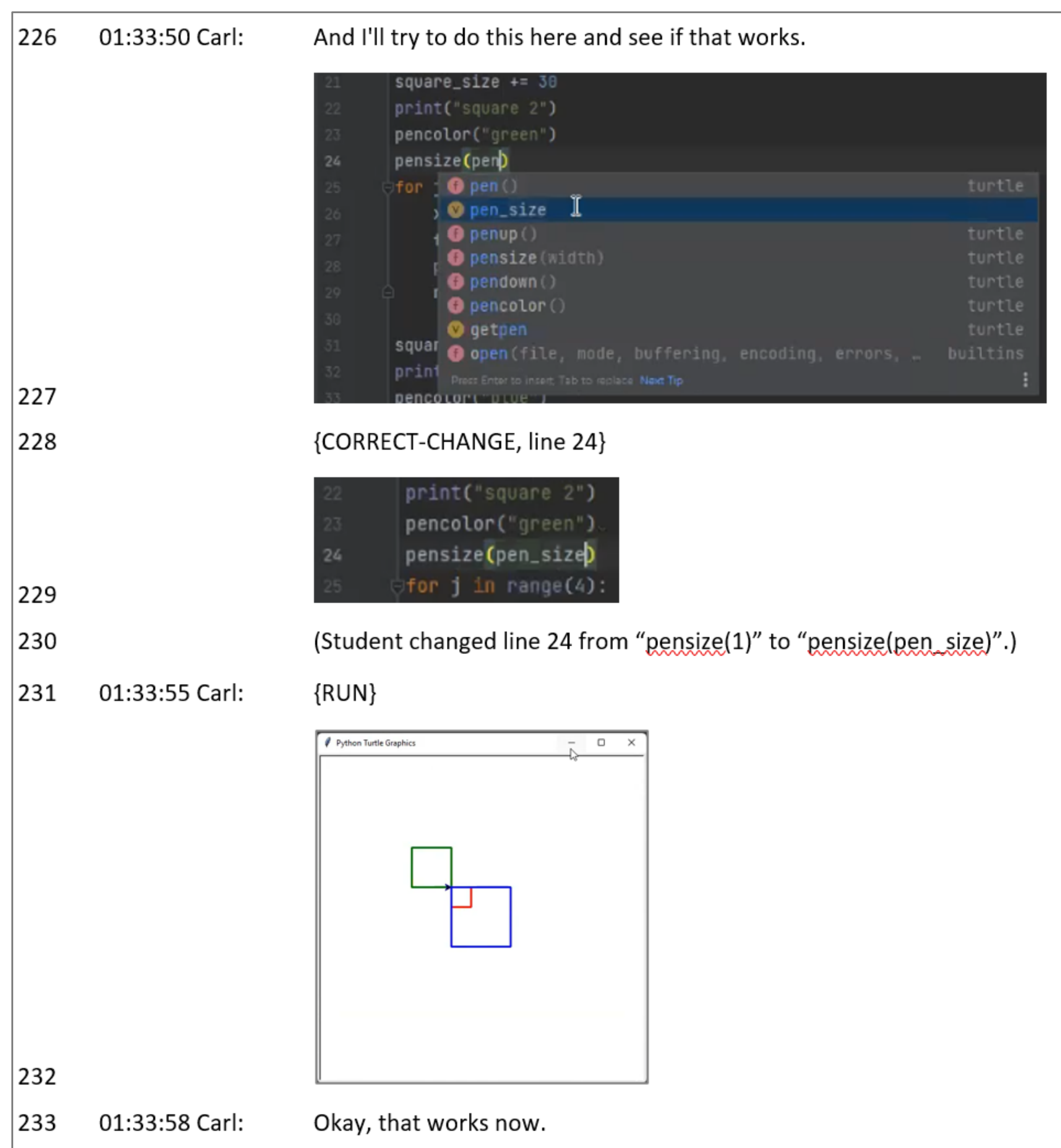
Two forms of data were collected during this study: video of student spoken words and actions, and student written responses to questions. The video was recorded using the Zoom video-conferencing software, while their responses to questions were either captured as spoken text or their written responses via the Zoom text chat feature. The video data collection and transcription are discussed in more detail below.

Participants performed their one-on-one debugging challenges individually, in a separate Zoom virtual breakout room. The audio and video for each session were recorded automatically by Zoom. Once in the breakout room, participants were asked to share their computer screen via the Zoom share screen feature. Once their screen was shared, they were given a URL to open in a web browser, asked to copy the relevant code from the resulting web page, create a new script in the PyCharm editor and name it to reflect the week of the challenge. They then copy and pasted the code from the web page into their editor and started to debug. Participants were reminded to speak aloud while they debugged.

At the end of each class (two hours), the video recording was downloaded from Zoom to a local video file. The local video files were then fed into a custom Python program written by the researcher that used the Whisper[19] speech recognition system API to convert the audio to text. The transcribed text was then reviewed and annotated manually by the researcher with screenshots from the video to create anonymized student debugging session transcripts (see Figure 4.13).

---

[19] OpenAI's Whisper automatic speech recognition (ASR) system. https://openai.com/research/whisper



226	01:33:50 Carl:	And I'll try to do this here and see if that works.
227		
228		{CORRECT-CHANGE, line 24}
229		
230		(Student changed line 24 from “pensize(1)” to “pensize(pen_size)”.)
231	01:33:55 Carl:	{RUN}
232		
233	01:33:58 Carl:	Okay, that works now.

Figure 4.13. Partial debugging session transcript for participant Carl.

To capture student actions and spoken thoughts during their one-on-one debugging sessions, a concurrent think-aloud verbal protocol was used along with screen recording, in line with previous debugging studies (Jeffries, 1982; Nanja & Cook, 1987; Vessey, 1984; Whalley et al., 2023). Participants were asked to describe what they were “thinking, feeling, seeing, and doing” as they debugged. When they were silent for too long or if they made an action without describing it, they were reminded to speak-aloud. Participants were reminded that they could withdraw or stop at any time, in

accordance with the ethics protocol for the study. No maximum time was pre-set for the sessions to avoid rushing or stopping students prematurely and to avoid potentially adding additional pressure or stress due to a time constraint.

4.9. Data Analysis

The debugging session data collection resulted in 24 individual debugging session transcripts, one for each student-week session. Each of the debugging sessions were then coded using a code book (see Appendix A) to generate "debugging episodes" as shown in Figure 4.14. Adopted from Vessey (1984), the debugging sessions are a step-by-step trace of the student's actions as they attempt to debug a challenge. Their purpose is to provide a coded representation of the actions taken which can be used to discover patterns of approaches used by the participants.

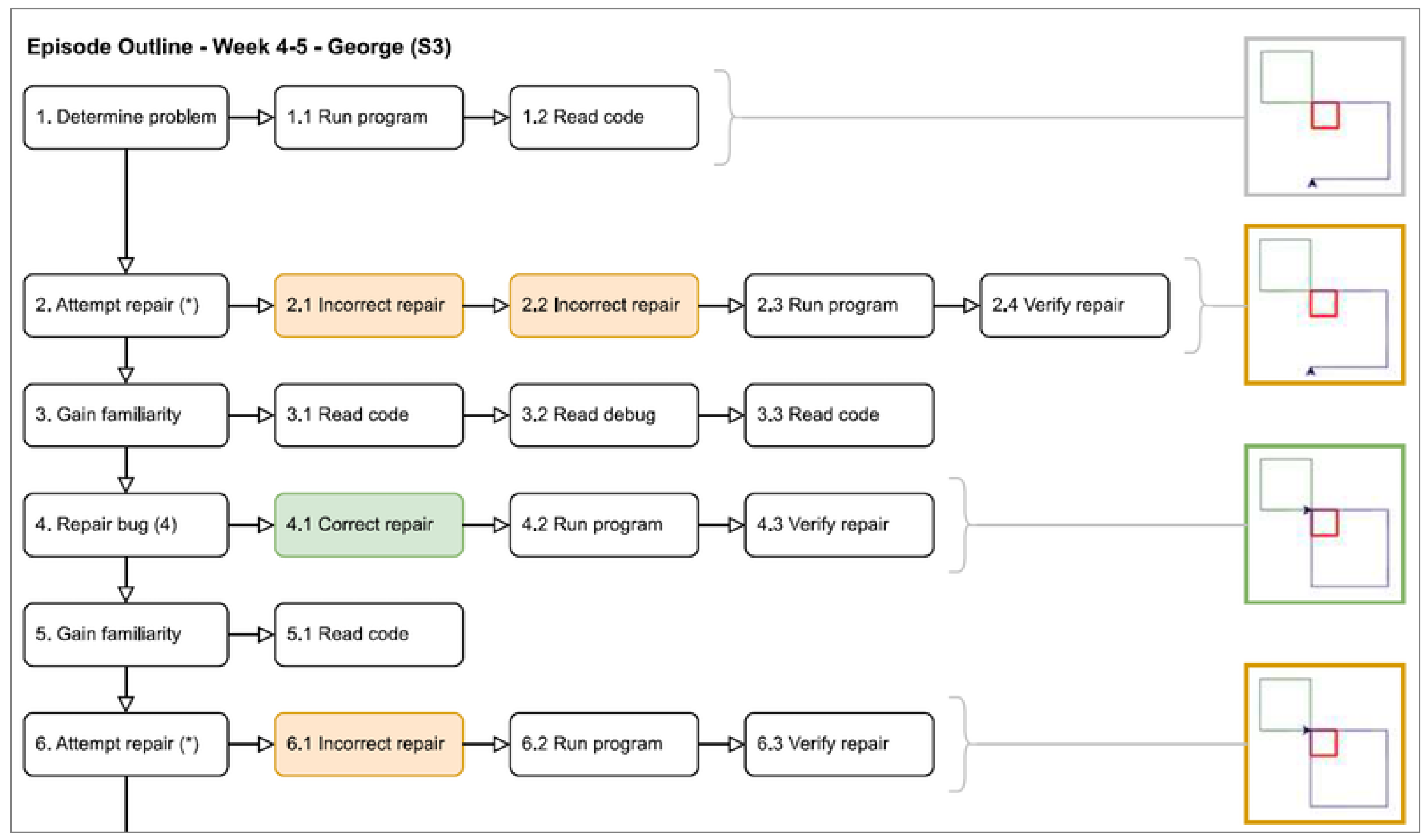


Figure 4.14. Sample debugging episode outline – Weeks 4/5 – George.

In order to create the debugging sessions, I first needed to create a code book that consisted of the codes necessary to describe the participant actions. Initially I started by inductively creating codes to describe the actions as I observed them within the sessions. However, after completing several passes of the transcribed debugging sessions and reviewing them with a second coder, I discovered that the inductively generated codes were not converging and required too much additional context for the second coder.

As a result of this lack of convergence of the codes, I went in search of an alternative coding mechanisms and discovered Vessey's Coder's Manual (1984, p. 421) developed for her debugging strategies study. Although our studies were similar, they also differed in a number of ways. For example, Vessey's study focused on adult professional programmers, used a different programming language, and participants performed all of their debugging offline using printed copies of the code rather than online using a modern programming editor. As a result, I extended and modified her codes to suit the context of this study and arrived at a mix of deductive codes from Vessey and inductively derived codes unique to the data within this study.

A second coder was used to improve the consistency of categorizations and validate the individual codes as suggested by multiple sources (Bowles, 2010; Ericsson & Simon, 1993; Hays & Singh, 2011; Miles & Huberman, 1994). Vessey (1984) accomplished this by employing two independent coders for her data while resource limitations for this study allowed only for a single second volunteer coder. As with Vessey's study, interrater reliability was measured using Cohen's kappa (Cohen, 1960). However, it should be noted that interrater reliability is not without criticism and controversy. For example, some argue that the need to reach agreement between multiple parties may reduce the interpretation to a lowest common denominator and thus drop or lose the richness of the original data. As well, adding additional coders adds more resource requirements to an already resource intensive process (Chiu & Shu, 2011). Although both criticisms are valid, the potential loss of richness of the data is still expressed when necessary and the additional resources needed was considered an acceptable cost for the consistency gained.

Chapter 5.

Findings

This chapter presents the data collected to describe the debugging strategies demonstrated and the challenges faced by upper elementary-aged novice Python programmers during an eight-week programming course that included explicit debugging instruction. This chapter starts with a review of the research questions, data sources, and participants to contextualize the data. Next, each of the weeks when data was collected, are discussed chronologically to illustrate the progression of debugging strategies and student attitudes over the course of the study. Each week covers the strategies and challenges observed, how long each student took to complete them, and student self-reported self-efficacy ratings. The chapter concludes with a summary of students' self-efficacy ratings and their descriptions of debugging over the course of the study.

5.1. Research Questions

This study addressed four research questions:

RQ1: What strategies do novice elementary school students employ when debugging Python code?

RQ2: What challenges do novice students face when debugging?

RQ3: What effects does an explicit debugging strategy have on student debugging strategies?

RQ4: In what ways do students respond to explicit debugging instruction?

5.2. Data Sources

There were two primary sources of data for this study: Zoom video recordings of one-on-one debugging sessions and student responses to questions answered via Zoom text chat. The debugging session recordings were transcribed to produce timed

action and statement data that were coded and described in this chapter. After four rounds of sampling and coding, 12% of the available samples had been randomly selected, coded, and then compared by both coders. By the fourth round, we achieved a Cohen's kappa of $\kappa = 0.7987$ for the individual codes and $\kappa = 0.8645$ for the summarized code categories.

Student responses to survey questions are also described in this chapter.

5.3. Participants

Note that pseudonyms have been substituted for actual participant names throughout this thesis.

The participants in this study were nine male[20] novice Python programmer upper-elementary students, aged 10-12 (Grades 5-7). Similar to Murphy et al. (Murphy et al., 2008), the novices in this study were not completely new to Python programming. Instead, students had all participated in at least one previous Python programming course with the researcher and in some cases up to four additional eight-week courses or camps. As well, all students had participated in at least one course online via Zoom with the researcher before, and so they were familiar with how to use Zoom in a classroom setting.

The rationale for studying students with some pre-existing introductory knowledge is that although they were still novice programmers, they had completed enough prior direct instruction in Python programming to be familiar with the basic syntax, and have debugged a variety of errors. It therefore seemed likely that they have been programming long enough to have established their own debugging strategies, even if only rudimentary in nature. Klahr & Carver noted that a group similar in age to this study's group, 15 fifth grade students (~10 years of age), even with 200 hours of unstructured Logo experience, failed to develop techniques for identifying and locating bugs on their own.

[20] Two female students and five additional male participants expressed interest but were ultimately unable to join due to other obligations.

5.4. Week 1: One-on-one Debugging Sessions

The first one-on-one debugging session occurred during Week 1 and consisted of four short challenges, with the first three involving syntax errors and the fourth a semantic error. There was no time limit on the challenges and students were not told how many bugs there were in the code provided. Three strategies were observed for the syntax errors: using editor hints, using prior programming knowledge, and using compiler error message comprehension. Four strategies were observed for the semantic error: reading and tracing code, comparing the provided code to working code, using prior programming knowledge, and removing unnecessary code. A complete listing of the code and included bugs is presented in Appendix B, and a summary of the strategies is presented in Table 5.1.

Table 5.1. Week 1 student debugging strategies

Strategy	Description
Using code editor hints	Look for and react to code editor hints to identify syntax errors while editing or reading the code.
Using prior programming knowledge or experience.	Identify bugs using prior experience of the same or similar bugs.
Using compiler error message comprehension.	Identify and locate bugs using information from compiler error messages.
Reading and tracing code.	Develop familiarity with basic functions and flow of the code. May lead to identification of bugs.
Comparing to working code.	Compare to known-good or working code to identify incorrect differences for repair.
Removing unnecessary code.	Remove duplicate or code deemed unnecessary.

5.4.1. Week 1 Student Debugging Strategies

One of the first bug identification strategies observed to be used by students during Week 1 was reacting to and using hints provided by the PyCharm code editor. PyCharm provided visual hints in the form of a red underline beneath syntax errors (Figure 5.1) and yellow highlighting for warnings such as possible unused code (Figure 5.2). As a result, after copying and pasting the buggy code into PyCharm, most students immediately moved their mouse cursors to any underlined or highlighted text. For example, during Week 1, Walter noted an editor hint on line 4 for challenge #4, began

circling it (Figure 5.2) and said, "*I see that this is underlined in yellow*[21]." (Week 1a: 1:05:05). Lucas followed a similar pattern for challenge #2. He immediately moved his cursor to line 3 after pasting the code into the editor and stated, "*I see [that] all of this…*" (Week 1: 2:26:22), then circled the underlined incorrect uppercase *PRINT* text, deleted it, and replaced it with the correct lowercase *print*. All of these students moved their mouse cursors to the highlighted lines immediately after pasting the code into the PyCharm editor and before they had either run the code to generate a compiler error or even had time to read or trace the code.

```
1   # Challenge #1
2   age = 1
3   age = Age + 1
4   print(a)
```

Figure 5.1. Screenshot of editor error highlights for challenge #1 (Week 1).
* The variable "Age" on line 3 is underlined in red (error) because it should not be capitalized. The variable "a" on line 4 is underlined in red because no variable "a" exists and instead it should be "age".

```
1   # Challenge #4
2   height = 2
3   height = height + 2
4   height + 2
5   # at the end, the height should be 6
6   print(height)
7
```

Figure 5.2. Screenshot of editor warning highlight for challenge #4 (Week 1).
* The text "height + 2" on line 4 is highlighted in yellow because it does not result in any change to the program state.

In addition to using code editor hints to locate potential errors, some students also combined the editor hint with syntax recognition, prior debugging experience, and program comprehension. For example, Carl, during challenge #2, immediately moved

[21] Walter stated *underlined* but meant *highlighted*.

his mouse to line 3, where the function *print* was incorrectly capitalized as *PRINT* and noted, "*Print is (mis)spelled with full capital, so it needs to be fully lowercase.*" (Week 1: 1:35:07). Peter made a similar statement during challenge #3 which involved an invalid leading space before the statement. While reading the code and before running it, Peter correctly noted, "*There shouldn't be spaces before the print because it's not in a loop or anything.*" (Week 1: 2:33:13). In both cases, the students used the code editor hints to locate the errors but then used their knowledge of the syntax or prior debugging knowledge to correct the error.

In addition to using hints from the editor and code comprehension to identify errors, students also used compiler and runtime error messages to both identify that there was an error and to locate it within the code. This was especially common for the first three challenges during Week 1, because students were instructed to run the code for the first challenge as soon as they pasted the copied code into their editor to ensure they were running the correct code. As a result, students experienced a compiler error message early in the debugging process as shown in Figure 5.3. For example, George upon seeing the *NameError* shown in Figure 5.3 stated, "*It's like a name error, which means age needs a definition…*" (Week 1: 1:43:40) and then noted the reported line number, "*Oh, it's line 3.*" (Week 1: 1:43:55). He then returned to the code, clicked on line 3 and after reading the line stated, "*Oh, I think this should be lowercase.*" (Week 1: 1:44:01). Other students ran the program, moved their mouse to the compiler error area at the bottom of the editor, and then immediately clicked on the line with the error, thus using the compiler error message output to both identify and locate the error.

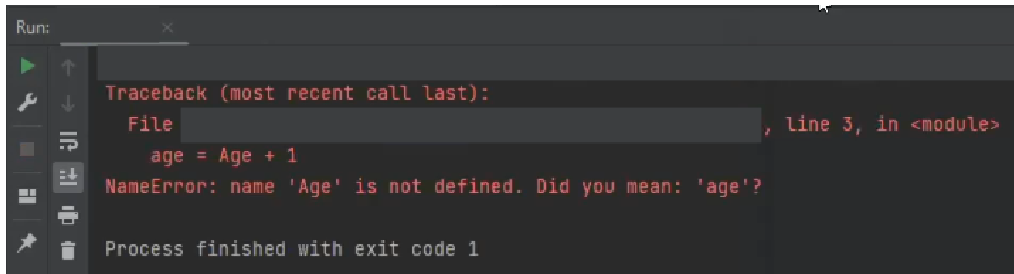


Figure 5.3. Screen of sample compiler error for challenge #1 (Week 1).
* The file name and path are redacted to hide student name and system information.

Unlike the approaches used to debug syntax and runtime errors, which relied heavily on editor hints and compiler error messages, debugging the semantic error in challenge #4 required the students to understand what the program was supposed to do

and then compare it to what it was actually doing. Most students started off by scanning the code for syntax errors or editor hints. For example, Oscar noted, "*Something in this… is off.*" (Week 1: 2:21:06) while circling line 4, which was yellow highlighted as shown in Figure 5.2. However, unlike the first three challenges, the program was syntactically valid and when run, the program would return the value 4 rather than a compiler error as happened with the previous three challenges. As a result, students had to read and trace the program to try and figure out what the program was actually doing and from there, four strategies emerged: reading and tracing the code, comparing to working code, using prior programming knowledge, and removing unnecessary code.

While reading and tracing challenge #4, most students correctly determined the initial value was *2*, based on line 2 (*height* = 2). They then noted the addition on line 3 (*height* = *height* + *2*) but were confused that line 4 (*height* + *2*) did not add two more and instead resulted in no change to the value of height. It should be noted that the editor highlighted the line in yellow and reported, "*Statement seems to have no effect*", however, none of the students hovered over the line to see the editor hint and none of the students added a debug print-statement, such as *print(height)*, to trace the code. Ideally, students should have at least read the editor hint or printed out the value of height to determine its value (something they would learn to do during subsequent classes during the course). Instead, they made guesses based on their reading and understanding of the code, and three strategies emerged.

The first strategy was to compare the bugged line 4 to the similar working code on line 3. For example, George noted that lines 3 and 4 were similar and stated, "*… should I do height equals height plus 2?*" (Week 1: 1:46:33). He then changed line 4 to match line 3 (*height* = *height* + *2*), and ran the program to verify the repair was correct. Jason, Lucas, Mark, Oscar, and Walter all arrived at the same solution. Peter made a different change, noting that the line was only missing an equal sign and correctly changed line 4 to *line* += *2*. When asked why, he correctly recalled a previous programming exercise and stated, "*Because plus equals (+=) is for adding things, and plus is kind of just for combining things, sort of.*" (Week 1: 02:33:58). Finally, Brian and Carl arrived at a different solution. Brian noted that line 4 was unnecessary or redundant, so he deleted the duplicate line and instead changed line 3 to add +4 to height's initial value of 2 to arrive at the correct final value of 6. Carl ultimately arrived at the same solution as Brian, but as is detailed in the challenges below, did not do so

immediately. Thus, all of the students arrived at the correct final value, but used different strategies to do so.

5.4.2. Week 1 Student Debugging Challenges

All of the strategies described in the previous section resulted in correct solutions, however, two observed strategies were counterproductive. The challenges are summarized in Table 5.2 and then described in more detail with student examples.

Table 5.2. Week 1 student debugging challenges

Challenge	Counterproductive use
Introducing new bugs.	Changing the code unintentionally introduces new bugs.
Getting stuck on a repair.	Vacillating when faced with a repair decision so much that they stop, shutdown, or give up.

Two common challenges were noted during Week 1: introducing new bugs into the code and getting stuck on a repair.

Typically during debugging, code has to be modified to correct bugs, a process called repairing. During these repairs, new bugs can be introduced either by adding incorrect new code, changing working code to no longer work, or removing working code. For example, Mark, in response to the compiler error caused by the variable *a* not being defined, incorrectly changed line 4 from printing the value of the variable *a* to printing the text *a* instead. Although he correctly identified the cause of the error, stating, "*There's no, what's it called, name A?*" (Week 1: 2:07:48), he then incorrectly noted, "*It has to be under quotations, doesn't it?*" (Week 1: 2:07:52) and incorrectly changed *print(a)* to *print("a")*. He then unexpectedly moved the newly modified *print("a")* to line 2, which resulted in no output of the final value of age. Upon running the program and seeing no output, he identified the mistake, deleted the quotations, and ended up with the correct *print(age)*, but on the wrong line. After seeing this, he copied his new *print(age)* back down to where the original *print(a)* was, ran the code, received the correct output, and moved on. Although Mark did arrive at the correct final answer in the end, his changes resulted in new bugs being introduced, one that remained and one that was corrected. During challenge #2 Mark made the same mistake, adding quotation marks around the variable to try to correct the reported syntax error.

Like Mark, Oscar unexpectedly introduced new bugs into his code as well. In Oscar's case, he changed the starting value for the variable age from one to seven, thus changing the final output from the expected value of *2* to *7*. However, unlike Mark who was trying to correct a syntax error, Oscar's change did not cause a syntax or runtime error and instead resulted in a subtle semantic error, because the final output of *7* did not match the target of *2*. Unfortunately, unlike challenge #4 which contained a code comment telling the students what the final value should be, challenge #1 did not. As a result, Oscar mistakenly changed the starting value of the age variable because he said that it seemed too low. This was not investigated further because the original bug was not related to the initial value of the variable but rather was ensuring the final value was output, even, as in this case, the value was different than expected.

Unlike Mark and Oscar, who successfully overcame their introductions of bugs while debugging, Carl got stuck during challenge #4, was not sure how to proceed, and gave up. Although Carl immediately identified the bugged line, "*Wait, hey, this (line #4) is supposed to be equals?*" (Week 1: 1:36:41), he was unsure of how to correct it, stating, "*I'm actually not sure.*" (Week 1: 1:36:49) Like some of the other students, he pointed out PyCharm's highlighting of the line, noting that line four was "*grayed out*" (see Figure 5.2). He also correctly identified that the line needed an equal sign (=) for assignment, but then mistakenly removed the addition operator (+) and replaced it with an equal sign (resulting in Figure 5.4). However, this change introduced a new error, like Mark and Oscar, and resulted in the final output being two rather than the four he started with and the six he was trying to achieve. After trying a few more combinations of equals and addition operators, running the program, and continuing to get incorrect results, Carl stated, "*I'm not sure.*" (Week 1: 1:38:36). He appeared stuck, repeating that he did not know what to do two more times. At this point the researcher suggested that he try something and see what happens. At that prompt, Carl deleted line 4 entirely and modified line 3 to add four to the initial value of height, to end up with the correct total of six. (See Figure 5.5 for Carl's ultimate solution.) During his post-challenge rating of self-confidence, Carl noted that the challenge had negatively affected his (self-)confidence.

```
1   # Challenge #4
2   height = 2
3   height = height + 2
4   height = 2
5   # at the end, the height should be 6
6   print(height)
7
```

**Figure 5.4.	Screenshot of Carl's edited code for challenge #4 (Week 1).**

```
1   # Challenge #4
2   height = 2
3   height = height + 4
4
5   # at the end, the height shoul
6   print(height)
```

**Figure 5.5.	Screenshot of Carl's solution to challenge #4 (Week 1).**
Carl deleted line 4 (height + 2) and instead modified line 3 to add 4 instead of 2.

5.4.3. Week 1 Debugging Times

In addition to the student strategies and challenges already discussed, all actions were implicitly timed as a result of being recorded during the debugging sessions. As a result, timing data for each of the students is presented here to show how long each participant took to complete each challenge. Unlike some of the previous studies that looked *only* at the time taken to debug (e.g., Gould & Drongowski, 1974), this timing data is meant to provide additional context rather than be the focus of the discussion. For example, one student struggled and took longer which resulted in a loss of confidence while another student also took longer and reported no impact on their confidence. In addition, the timing data provides some insights into the impact of student strategy choices, especially when counterproductive strategies resulted in longer overall times.

During Week 1, each student's time to complete each challenge was recorded. For Week 1's debugging challenges, there were nine participants, who took a minimum time of 1:54 to solve all four challenges and a maximum time of 5:28 (mean: 3:22, median: 3:21, SD: 1:09). Figure 5.6, shows the times for each student. Details for some of the times follow.

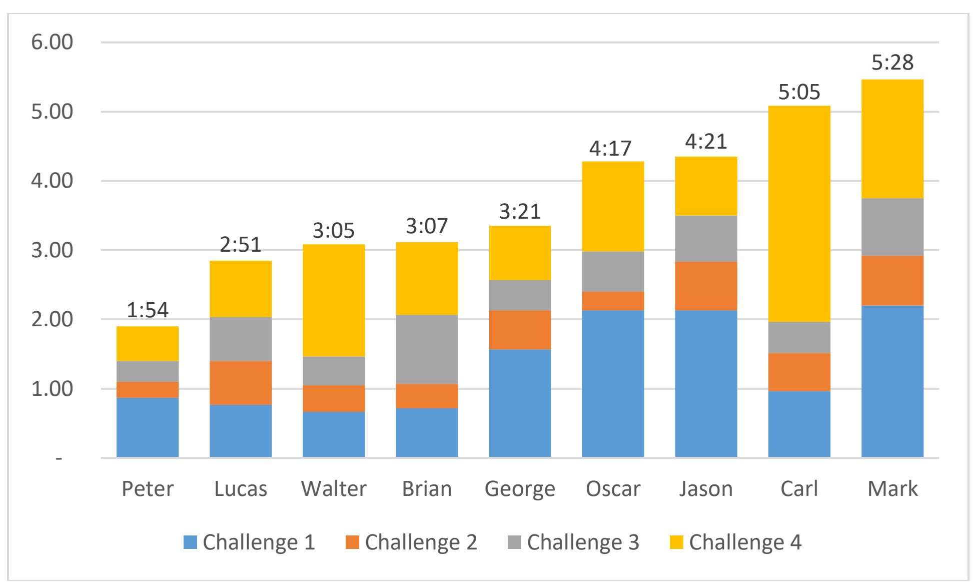


Figure 5.6. Week 1 debugging challenge times (in minutes, sorted by time).

For the first challenge, students were required to correct the capitalization for the variable *age* on line 3 and then print the value of *age* on line 4. Oscar immediately noted the red underline for the incorrectly named variable *Age* on line 3 and corrected it. However, he failed to recognize the invalid variable name *a* on line 4 as a typographical error and instead deleted it, resulting in line 4 printing a blank line instead of the value for *age*. Although this now resulted in zero errors, it also resulted in no output being produced. When asked if he had printed age, as required, he said no and then unexpectedly deleted the value 1 on line 2 (*age* = *1*) and replaced it with 7, resulting in a new starting age value (*age* = *7*). He then ran the code again and the output remained blank. Seeing no output, he then changed line 4 to *print(7)*, ran the program again, and correctly noted that the age should have been higher because of the *age* = *age* + *1* on line 3. Finally, he realized that he needed to print the variable *age* rather than the constant *7*, and corrected line 4. Altogether, his missteps resulted in a total debugging time for challenge #1 of 2:08.

Although Jason had the same solution time as Oscar for challenge #1 (2:08), his actions and process were completely different. Jason started by carefully reading through the code looking for clues. He correctly identified the errors, such as line 3, stating, "… *in one of the red lines, it said age equal age capital plus 1.*" (Week 1: 1:58:08) and line 4, "*… alpha print a, it needs to be age, not a.*" (Week 1: 1:58:34). However, he did not make any changes to the code immediately. It was not until he was told that he could make changes by the researcher, forty seconds into the challenge, that he made the first of his two correct changes. However, again, he paused after making the changes and it was not until the researcher asked if he was getting the correct answer that he acknowledged that he had and that he was done (16 seconds). Therefore, although his time was the same as Oscar's (2:08), much of his time was the result of him hesitating before making the changes he had already identified as needed.

Mark, like Oscar, added extra code and modified the program during challenge #1 resulting in him adding extra work and thus taking extra time. In Mark's case, he mistakenly changed the print on line 4 to output the text string *a* instead of the value of the variable *a*. He also added extra print statements once he realized he was required to output the value for the variable *age*. The addition of these extra print statements resulted in him undertaking unnecessary additional work, and thus resulted in a longer debugging time of 2:12 for challenge #1.

Carl, as previously mentioned, struggled with challenge #4, resulting in the highest overall solution time of 3:07 (mean: 1:18, median: 1:03, SD: 0:47). Although he correctly identified where the error was on line 3, he was not initially sure how to correct it, had long pauses with no activity, and tried a few changes unsuccessfully until he finally deleted the problematic line and changed line 2 instead to add the necessary value of four. Carl, like Jason, was hesitant to make changes, but unlike Jason made a number of incorrect changes before stopping and needing to be encouraged to make the final necessary change to complete the challenge.

5.4.4. Week 1 Pre/Post Self-Efficacy Ratings

As part of the one-on-one debugging sessions, students were asked to rate their level of confidence debugging Python code. These self-reported ratings were meant to provide insight into the impact of the challenges on their self-confidence. For example, if

a student struggled to debug a problem, it was assumed that they might report a reduced self-confidence score after the event. Additionally, to supplement the student rating values, students were also asked to explain why their score increased, decreased, or stayed the same. This additional question was meant to provide context for the changes beyond the numeric change.

Before and after completing the Week 1 debugging challenge, students were asked to rate their level of confidence debugging Python code on a scale from one to seven. See Table 5.3.

Table 5.3. Week 1 student self-reported self-efficacy ratings

Student	Pre-	Post-	Diff.	Student answer to why pre- and post- difference.
Brian	4.5	5.0	+0.5	*Exercises made me feel more confident.*
Carl	5.0	3.5	*-1.5*	*I kind of overthink things and that lowers my confidence.*
George	2.0	3.0	+1.0	*I was expecting it to be more difficult.*
Jason	5.0	6.0	+1.0	*I did all of this, and I felt confident about it.*
Lucas	5.0	6.0	+1.0	*Kinda easy if you know code already. I just got a bit confident.*
Mark	3.5	5.0	+1.5	*I think I wasn't sure, but I feel better.*
Oscar	4.5	5.5	+1.0	*I honestly didn't think I could do that.*
Peter	5.0	5.0	-	*It was a pretty accurate guess.*
Walter	5.0	5.0	-	*Sometimes I got it wrong, most of the time I got it right.*

All of the students successfully completed all of the challenges. Most students (6/9) reported that successfully completing the challenges made them feel more confident, and thus reported a higher post-activity rating. Of those six, three (Jason, Mark, Oscar) reported that they had been uncertain at the start but after completing the challenges they felt more confident in their abilities. Two students (George, Lucas) reported that the challenges had been easy, and their statements aligned with their challenge completion times as well. Two students (Peter, Walter) reported no change in their level of confidence, with Peter reporting that he felt his initial rating was already accurate.

Carl was the only student to report a lower post-challenge score and was the only student to really struggle with any of the challenges. In Carl’s case, during challenge (#4), he identified the general problem immediately but was unsure what repair to make. As a result, he vacillated between a few choices, started, stopped,

undid, and was indecisive. After almost giving up, I encouraged him to try one of his ideas and see what happens. He did so, by deleting the problem line altogether and fixed the code successfully. After successfully completing the problem and asked why he rated his confidence lower, he said that he tended to overthink problems some of the time.

5.5. Week 2: Student-Generated Examples and Expert Debugging

In an effort to gauge students' impressions of an explicit debugging strategy, they were introduced to an expert debugging process during Week 2 and asked to describe the difference between their own debugging approach and the one demonstrated. To start, students were asked to create a copy of their in-class exercise and then introduce up to three bugs into their previously working code. They were then asked to share their screen with the researcher and demonstrate what the output from their non-bugged version looked like. After showing their non-bugged version, they then sent the bugged version to the researcher. Then, the researcher debugged their bugged version while thinking aloud. At the end of this debugging demonstration, students were asked to provide a short one or two sentence description of the differences they noticed between the researcher's debugging process and their own.

Two students reported seeing no difference (Carl, Peter). Two students noted that the researcher ran the program after every fix (Mark, Walter), and Brian added that he did too but only after fixing multiple errors. Six of the students noted that the researcher worked through the process in order (systematically), one error at a time, and repeated the same process each time. Oscar was the only student who did not provide a reflective answer. His answer implied that the only important difference between his debugging approach and the researcher's was that the researcher had more experience and was "better at coding". Table 5.4 displays all of the students' responses to the question about how their personal debugging approach differed from that of the researcher.

Table 5.4. Week 2 students' observations of differences between the teacher (expert) and their debugging techniques (novices)

Student	Student description of differences between expert debugging and their debugging.
Brian	*You always try running it before debugging it while if I see an error, I will fix it without running it to see what it says. Well, I still run it but just after I fix it.*
Carl	*I couldn't really tell the difference.*
George	*You go through all the possibilities of errors in the code.*
Jason	*You do everything in order, and I find a mistake and fix it.*
Lucas	*I found that you took more time to find the answers.*
Mark	*You checked errors to make sure they worked before running and ran the code often.*
Oscar	*You are better at coding.*
Peter	*I didn't see a difference except you're faster.*
Walter	*You run every time once you think you have fixed the bug.*

5.6. Weeks 4/5: One-on-one Debugging Challenges

Unlike Week 1, which focused primarily on syntax errors, the challenges for Weeks 4/5 contained four semantic errors that required the students to identify mismatches between the target output in the specification and their actual observed output. The debugging challenge itself was a series of three squares in a particular order, colour, size, and thickness, as shown in Figure 3.11. There was no time limit on the challenge, and students were not told how many bugs there were -- only what the code should output, and what the code actually output. As a result of the shift towards identifying visual output differences, the challenge was modified so that students were first shown the target and actual outputs and then asked to describe what they saw and speculate on the causes before being given the code to debug. A complete listing of the code and bugs is provided in Appendix B, and a description of the bugs can be found in Chapter 4.

5.6.1. Weeks 4/5 Student Debugging Strategies

During Weeks 4/5, students demonstrated the following strategies: running the program first, sequentially reading and tracing the code, comparing similar code to spot differences, guessing-and-checking changes, working around the bug, and using debug print-statement output to understand what was happening inside the code. A complete list of the strategies and fuller descriptions are provided in Table 5.5.

Table 5.5. Weeks 4/5 student debugging strategies

Strategy	Description
Reading and tracing code.	Develop familiarity with basic functions and flow of the code. May lead to identification of bugs.
Running program first.	May generate compiler error messages, runtime error messages, or produces the output for comparison to the target.
Comparing to working code.	Compare to known-good or working code to identify incorrect differences for repair.
Guessing-and-checking changes.	After identifying and locating a potential repair, making a repair (guess), and verifying (checks) whether the repair is valid or not. Undoing the repair if it is not valid.
Adding missing code.	Adding missing necessary code.
Removing unnecessary code.	Remove duplicate or code deemed unnecessary.
Using debug print-statements.	Helps verify how the code executes and outputs the program state during execution. Can help with tracing and segmenting the code.

Most of the students (6/9; Brian, George, Jason, Lucas, Mark, & Walter) started the debugging challenge for Weeks 4/5 by running the program first. Unlike Week 1, where students ran the code for their first challenge because they were instructed to, during Weeks 4/5 most students chose to do this without prompting. The researcher employed this run-first strategy during their expert debugging demonstration during Week 2 as a way of both checking the output and identifying different kinds of errors. Only one student (Mark) commented about running the program first, stating that he wanted to see what happened as he traced the lines with his mouse as they were drawn. None of the other students provided a reason for choosing to run their program first, and most just closed the turtle window and jumped directly into the code once it finished drawing. Although the students did not provide an explicit reason for why they just closed the turtle window after running the code, it is probable that they were looking for (or testing for) compiler or runtime errors as were common during Week 1. However, because the Weeks 4/5 program did not have any syntax or runtime errors, the program ran first time instead.

All of the students started by slowly reading the code from top to bottom. This behaviour was more noticeable during Weeks 4/5 because the program was 41 lines long – much longer than the code for previous challenges which averaged only four

lines. As a result, students had to scroll the editor's text window up and down to see all of the code.

The code for Weeks 4/5 was grouped into three sections of repeated code blocks, one for each of the three squares, as shown in Figure 4.11. Students demonstrated an understanding that the code for the three squares was similar by comparing the code for each of the three squares to each other by scrolling up and down in the code or mentioning the similarities between the three code blocks. For example, George, while comparing code for squares one and two noted, "*Oh, I think I found it.*" (Week 5: 1:18;34) and then, "*I think there's a minus sign here.*" (Week 5: 1:18:45). He then correctly removed the minus sign and ran the program to verify the change. Walter made a similar observation, stating, "*I feel like because it says pen size one (line 24) and one (line 34), I feel like this should be pen underscore size (like line 14).*" (Week 5: 1:56:57), whereupon he copied the correct variable *pen_size* from line 14 onto lines 24 and 34 to correct squares two and three. Most students demonstrated this approach, scrolling the code up and down as they compared one block of code with another.

Whereas students like Peter and Brian picked specific targets to debug and stated possible causes before fixing them, others like Lucas, Mark, and Oscar took a guess-and-check approach. In Oscar's case[22], he started with a slow sequential read of the code, remarked that there were no errors with each line, and then circled back to the top to start reading again, stating, "*I feel like there's something wrong. I'm just not catching it.*" (Week 4: 1:41:51). He then jumped around the code, clicking on various lines in different areas before changing the angle in the loop for square two from the correct *right(90)* to the incorrect *right(120)* and then running the program to see what impact this had. His first change resulted in an incorrect rotation, and so he immediately undid the change and replaced the correct *right(90)* with an incorrect *left(90)*. After running it, he remarked, "*Okay, that's makes it a bit better.*" (Week 4: 1:43:09), but then deleted *left(90)* and replaced it with *forward(50)* instead. Oscar continued making single changes (guesses) and running the program (checks) each time to see the results of his changes. He continued this approach several more times until a change resulted in a particularly unexpected output, whereupon he remarked, "*Oh, what. What is that?*" (Week 4: 1:44:11). After making no progress with these single changes, he finally

---

[22] Oscar's full episode trace for Weeks 4/5 can be found in Appendix D, Figure D8.

noticed the extra minus sign for square two while comparing the code for squares one and two, stating, "*Oh, this is x equals minus square.*" (Week 4: 01:45:03). Upon spotting this difference, he undid one of his previous changes, correctly removed the extra minus sign bug, and ran the code to see that square two was rotated correctly. Upon seeing the correction, he proceeded to undo other incorrect changes he had introduced, ran the code again and then quickly fixed the other three bugs. He did not initially identify that the code for the squares was repeated, and instead he repeated changed lines randomly to see what the change would do before discovering the similarities and using code comparison to find and fix the original bug.

Like Oscar, Lucas and Mark repeatedly guessed and checked changes; but unlike Oscar, they did not ultimately use a code comparison approach to identify and correct the bugs. Instead, they added new code that worked around the original bugs. Lucas started by adding an extra forward statement to fix the missing fourth line for square three, ran the program, saw only a small change, increased the value for his incorrect forward statement and ran the program again. Mark also added an extra forward statement for square three after the for-loop that replicated the code within the for-loop to add the missing fourth line. However, Mark took this further by rotating square two 180 degrees using *right(180)* so that it was in the correct position, at the cost of also rotating the third square. This required him to add a second *right(180)* rotation after square two to rotate square three as well. Although both students arrived at turtle outputs that matched the target, they did so by working around the original bugs rather than identifying and fixing the cause of the bugs. These kludgy solutions however are problematic because they make the code harder to read due to unnecessary extra steps and they fail to fix the underlying problem which means that it is still there to potentially cause problems in the future. As well, although not explored in this study, these suboptimal solutions could represent a pattern or misunderstanding that could be hindering their debugging or programming abilities. In Mark's case, he did note after the session that he could have fixed the missing fourth line for square three by fixing the range value, but chose not to, which implies that he was potentially aware of the work around.

Although the code for the Weeks 4/5 challenge included debug print-statements that output clues to what the program was doing, only Walter used them. After running the program for the first time, Walter read through the resulting debug output and noted

that square two had a negative rotation, stating, "*Oh, it's minus.*" (Week 5: 1:54:47). Upon discovering this, he compared it to the output from the other two squares and stated, "*So, I just need to figure out (where the minus is)…*" (Week 5: 1:54:58) and returned to the code in search of the minus sign that was causing the bug. However, he started at the top of the code, and while sequentially reading the code mistakenly identified the first minus sign in the code, which was in front of the x-parameter for the goto statement at the top of the code, i.e., *goto(-50, 50)*, and deleted it. After making this incorrect change, he ran the program and immediately saw that this was incorrect. He undid the incorrect change and restarted his sequential search through the code for another minus sign. After not finding another minus sign right away, he moved to the code for square two, selected it with his mouse and stated, "*So, it's something in here.*" (Week 5: 1:55:17). He then read the lines aloud sequentially until he found *x = -square_size* on line 36, whereupon he stated, "*Oh, there it is.*" (Week 5: 1:55:48). So, although Walter identified the correct issue, that the output was negative and that this was most likely caused by a minus sign, he did not immediately connect the bug to the correct location. Instead, he focused on finding the minus sign and incorrectly used a sequential search to find the first minus sign instead of the correct minus sign. However, after his initial mistake, he did jump to the correct area within the code and again applied a sequential search through the code for the error, reading the code aloud until he found the bug. At no point did he jump down to the debug print-statement for square two and compare the variable that was showing the negative value (x) to help locate the source of the error itself.

5.6.2. Weeks 4/5 Student Debugging Challenges

Similar to Week 1, some students during Weeks 4/5 demonstrated counterproductive or misapplied approaches including introducing new errors and adding unnecessary code. However, during Weeks 4/5, students exhibited additional challenges including: excessive guessing-and-checking and avoiding rather than fixing errors.

Table 5.6. Weeks 4/5 student debugging challenges

Challenge	Counterproductive use
Making excessive changes.	Repeatedly changing the same line or block of code rather than stepping back to consider alternatives or verifying that the code in question is the correct code to change.
Avoiding bugs.	Changing the code in such a way as to work around a bug, such that the original bug is not repaired and is avoided, worked around, or compensated for instead.

As noted in the previous strategies section, three of the students (Oscar, Lucas, and Mark) exhibited a pattern of guessing-and-checking. Although all students were ultimately successful and undid all of the incorrect changes they made, Oscar made significantly more changes (7) than the other students (mean: 2.22; SD: 2.17) and ran the program significantly more often while searching for bugs (14 times; mean: 7.7; SD: 4.4). Unlike Jason for example, who carefully read through the code until he identified where the code for the three squares was, Oscar chose to change an angle in the middle of the code to try to determine which square the code he was changing belonged to. He did this by repeatedly changing single statements, running the code, looking at the output, and then undoing the incorrect change. He repeated this process until he determined that the rotation was caused by the minus sign and not the numerous rotations or forward movements.

As also previously mentioned, Lucas and Mark both avoided or worked around the root causes of the bugs during the challenge for Weeks 4/5. In Mark's case, he correctly identified that square two was incorrectly rotated 180 degrees around the x-y axis, and so he rotated the square by 180 degrees. However, upon seeing that this repair caused the third square also to rotate 180 degrees, he again applied the same change to the third square to correct its orientation. In keeping with this approach of working around the problem, both Mark and Lucas misidentified the bug for square three, and rather than fixing the incorrect range value, they added extra code to add a single extra line. In Mark's case, he added extra code to duplicate the actions within the for-loop, but later remarked that he could have fixed the bug by changing the range value for-loop, but chose not to. In all cases, the students chose to change the code before correctly identifying what the bug was and as a result, once they had found a workable solution, they kept it so long as their output was correct, even if it did not

correct the underlying bug. This approach is problematic not only because it potentially leaves an unfixed bug and added extra unnecessary code, but it also means that the code is harder for others to read and understand which means that it will be harder to improve and maintain.

5.6.3. Weeks 4/5 Debugging Times

During Weeks 4/5 (Figure 5.7), there were nine participants. The minimum time a participant took to solve a challenge was 1:43 (Peter), the maximum was 21:37 (Jason), and an average time of 8:46 (median: 7:40, SD: 5:27). Jason's very long session (more than double the length of the next-longest one) skewed the average time considerably.

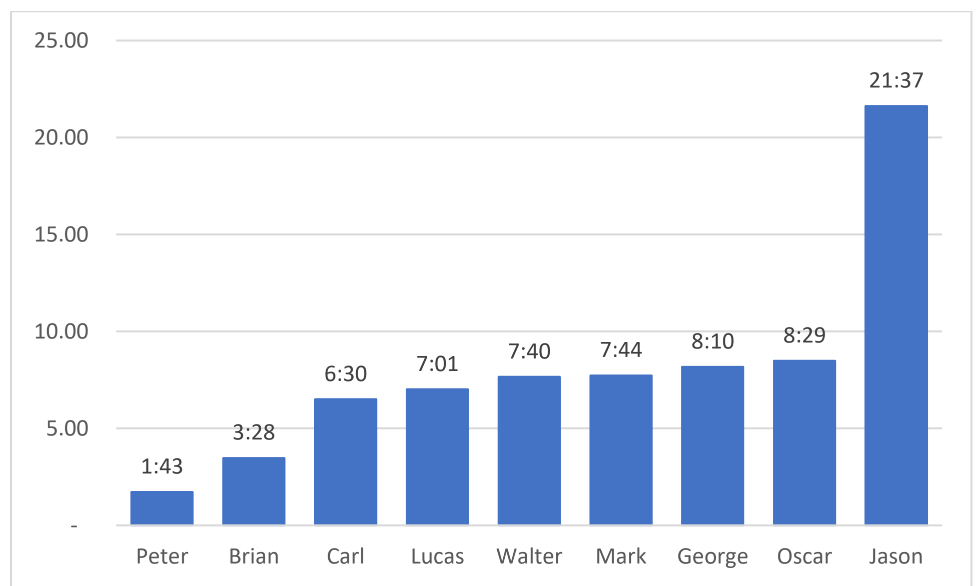


Figure 5.7. Weeks 4/5 debugging challenge times (in minutes, sorted by time).

Peter's session exhibited the most advanced approach to debugging strategy. As he read through the code, he identified and corrected each of the bugs in order, using his observations from the initial identification phase to guide his decisions. He corrected the rotation bug by noting the incorrect negative sign. He corrected the incomplete third square by changing the range of the loop to match the range in the other two squares. Finally, he correctly replaced the constant values of *1* for the incorrect pen thicknesses with the correct *pen_size* variable. As shown in his episode trace (Figure 5.8) his path was deliberate, and he only ran the code once at the end after correcting all of the bugs.

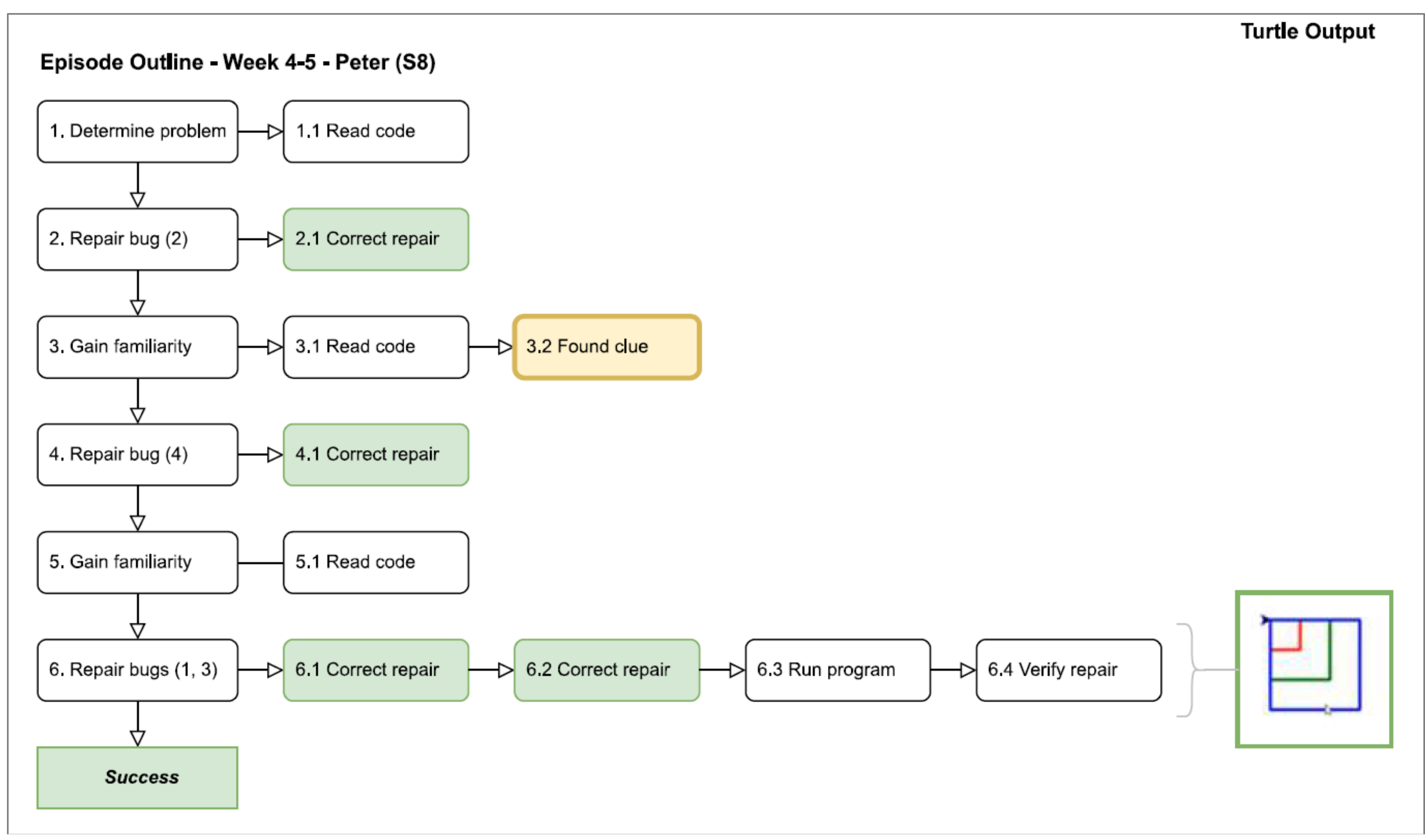


Figure 5.8. Peter's Week 4/5 debugging challenge episode trace.

It is useful to compare Peter's approach with that of Jason, who took the longest time to complete the challenge. While Peter acted decisively as he traced through the code, Jason was hesitant and repeatedly expressed uncertainty. For example, when asked where he wanted to start, he responded with, "*Maybe here.*" (Week 4: 1:21:32) and pointed at the output window within PyCharm. After reading through the initial output, he ran the program again without making any changes. He then chose to focus on why the third square was only drawing three of its four sides. He focused on the incorrect range value, changed it, and then ran the program again to see if the change was indeed correct or not. He then shifted focus to the incorrectly rotated square two. After reviewing the code, he correctly deleted the minus sign and reported that, "*… there's no such thing as minus square size.*" (Week 4: 1:29:25). While doing this, he scrolled the code up and down so he could compare the already-correct square_size variable for the previous square one that he had matched to repair the error for square two. Unlike Peter who corrected all of the bugs before running their program, Jason ran his program after every change, which was more similar to the behaviour demonstrated by the researcher during the Week 2 debugging examples. As well, Jason was deliberate and studied the code closely which resulted in long pauses, repeated runs of the code to double-check, and then slow careful changes, all of which added up to his longer time.

All episode traces for Weeks 4/5 are available in Appendix D.

5.6.4. Weeks 4/5 Pre/Post Self-Efficacy Ratings

Before and after completing the Weeks 4/5 debugging challenge, students were asked to rate their level of confidence debugging Python code on a scale from one to seven. Then, following the challenge, they were asked to provide a reason for any change in their rating. Individual participants' ratings with their reported reasons for changes are shown in Table 5.7.

Table 5.7. Weeks 4/5 student self-reported self-efficacy ratings

Student	Pre-	Post-	Diff.	Student answer to why pre- and post- difference.
Brian	5.0	5.0	-	*I don't know. It didn't feel like it changed.*
Carl	6.5	6.5	-	*Same. I think this is kind of how well I thought it would go.*
George	5.0	6.5	+1.5	*I don't know, but I felt better at the end.*
Jason	5.0	6.0	+1.0	*I felt it was kind of hard, and I solved it.*
Lucas	7.0	*9.0**	*+2.0*	*When I started noticing more stuff, I got more confident.*
Mark	3.5	4.5	+1.0	*Like the last time, I wasn't completely sure at how good I was... I might have raised it higher, but I spent too long at the start.*
Oscar	5.0	6.0	+1.0	*I honestly didn't think I wasn't going to do that… so this boosted my confidence.*
Peter	5.5	5.5	-	*Because I think I was able to find the errors quite fast, but not amazingly fast.*
Walter	5.5	6.0	+0.5	*Cause I feel like… before I would always go to you (teacher), but now I feel more comfortable/confident.*

* Lucas asked if he could choose a score of 9.0 out of 7.0 because they started with 7.0 and they felt more confident at the end of the challenge despite choosing an initial value of 7.0.

Of the students who reported an increase after successfully completing the challenge (6/9), most reported that they thought the problem was hard and that successfully debugging it boosted their confidence. Although Lucas reported the highest increase in confidence, he reported a score of 7.0 out of 7.0 at the start, and as a result was already at the maximum score. He therefore chose 9.0 out of 7.0 because he felt more confident after successfully completing the challenge. Four students (Brian, Carl, Lucas, Peter) reported no change in their confidence, and no students noted a drop in confidence -- not even Jason who took the most time to solve the problem. In fact, Jason reported an *increase* in his confidence because although he found the challenge hard, he was ultimately successful in solving it. Of particular note, Carl reported a consistent score of 6.5, which was significantly higher than his previous pre-score of 5.0 and post-score of 3.5 at the end of the previous challenges during Week 1.

5.7. Weeks 7/8: One-on-one Debugging Challenge

The final debugging challenge for this study, conducted during Weeks 7/8, involved two bugs, both of which were semantic bugs. One bug was a logical error, and the other an incorrectly commented-out line of code. The goal with this challenge was to test students' bug-locating skills by requiring them to trace the bug using the debug print-

statements provided to them. There was no time limit on the challenge, and students were not told how many bugs there were, only what the code should output and what the code actually output. As already noted, three students, Carl, Jason, and Walter did not participate during Weeks 7 and 8 because we ran out of time. A complete listing of the code and bugs is shown in Appendix B and a description of the bugs can be found in Chapter 4.

The focus of the Weeks 7/8 debugging challenge was to observe the extent to which students had adopted the explicit debugging techniques taught during the course. Of particular interest was their use of the debug print-statements for debugging state issues, as presented by the first bug, as well as their adherence to their previous ad-hoc debugging approaches, i.e., would they be systematic in their approach or not. In summary, two students did not use debug print-statements but did successfully debug the code. Two students did use debug print-statements to successfully debug the code but took much longer to do so and required some assistance. Finally, two students did not use the debug print-statements and were unsuccessful in completing the debugging challenge.

5.7.1. Weeks 7/8 Student Debugging Strategies

Table 5.8. Weeks 7/8 student debugging strategies

Strategy	Description
Using prior programming knowledge or experience.	Identify bugs using prior experience of the same or similar bugs.
Reading and tracing code.	Develop familiarity with basic functions and flow of the code. May lead to identification of bugs.
Running program first.	Generates compiler error messages, runtime error messages, and produces the output for comparison to the target.
Guessing-and-checking changes.	After identifying and locating a potential repair, making a repair (guess), and verifying (checks) whether the repair is valid or not. Undoing the repair if it is not valid.
Adding missing code.	Adding missing necessary code.
Removing unnecessary code.	Remove duplicate or code deemed unnecessary.
Using debug print-statements.	Helps verify how the code executes and outputs the program state during execution. Can help with tracing and segmenting the code.

As previously mentioned during Weeks 4/5, all students continued to read the code for Weeks 7/8 sequentially, despite the code at the top of the program being functions that did not contain any bugs. This meant that they unnecessarily read through all of the functions and set up code at the top of the program before eventually reading the main loop near the bottom of the code.

Of the two students who did not use the debug print-statements, Peter was the only student to explicitly note that the odd squares were coloured, and the even squares were not. As a result, although he read through the code sequentially, when he reached the main drawing loop, he paused and traced the code for drawing the squares more closely. After reading the main drawing loop closely, he correctly swapped two lines of code in the loop so that the even colours were white and the odd correctly used colours from the *colours_list* variable. Peter's process was exclusively focused on reading the code closely, simulating it in his head, and only running the program to verify his changes. Peter made no mistakes, did not use any of the included debug print-statements, did not introduce any new errors and completed the challenge soonest in 3:20 (see Figure 5.9).

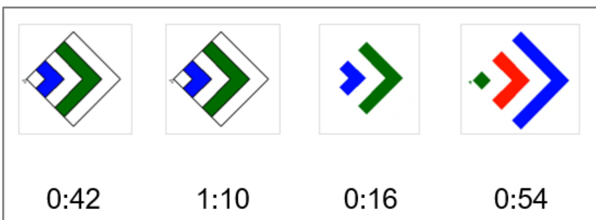


Figure 5.9. Peter's Week 8 debugging challenge turtle output progression.
Times shown below turtle output images are relative to the start of the program for the first image and then relative to the previous image. They are meant to show the passage of time between runs of the program that produced the turtle output.

Brian made the same change as Peter, but rather than changing the code manually, he swapped the code by dragging and dropping it between the True and False clauses of the if-statement. However, unlike Peter, Brian uncommented all of the debug print-statements at the start of his session during his initial read of the code. After doing that and running the program, he read through the debug output once but made no mention of it again. Brian's process, like Peter's, involved careful reading and simulating of the code. While debugging, he introduced and undid four errors, ran the code ten times, and completed the challenge in 8:57 (see Figure 5.10).

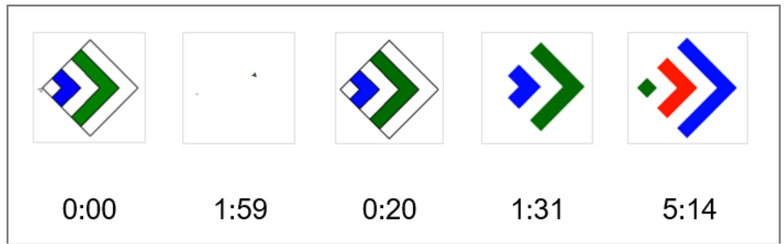


Figure 5.10. Brian's Week 8 debugging challenge turtle output progression.

Although Peter and Brian did ultimately produce correct output, they did so by accepting that the *is_even* function returned a semantically incorrect result and worked with the function as-is rather than correcting the function itself (which is the where the bug was). Had the *is_even* function been used in other places within the program, thus resulting in multiple bugs because of the errant function, they would have been forced to make repeated fixes to the code to compensate for the bug in the function. In contrast, the next two students eventually identified the problem in the *is_even* function and changed *it* to correct the problem instead.

The two students who successfully completed the challenge while using debug print-statements were George and Mark. George started off by adding his own debug print-statement to determine the order of execution of the functions. He was the only student who explicitly attempted to use the Wolf-Fence algorithm shown in class to locate the bug within the code. Following this single use of the algorithm, he determined the general area of the code where the bug was, but did not repeat the process to further narrow the search as prescribed by the algorithm. After a number of attempted changes to one of the modulo operators for the colours, he uncommented the provided debug print-statements, initially all at once but then gradually one-by-one, to see what information they provided. It was while reading the individual debug output lines shown in Figure 5.12, that he noted, "… *is_even five true*…" (Week 8: 2:11:59) followed by, "*Shouldn't it always be true?*" (Week 8: 2:12:02), and then exclaimed correctly, "*Oh, I have to flip this around.*" (Week 8: 2:12:10). However, rather than immediately doing so, he tried a few more changes of the modulo operator before returning to focus on the *is_even* function. Once focused on the *is_even* function, he correctly identified and then switched the return values to return True for even numbers rather than odd numbers. George introduced and undid nine errors, ran the program 26 times, added six debug print-statements, and completed the challenge in 17:04 (see Figure 5.11).

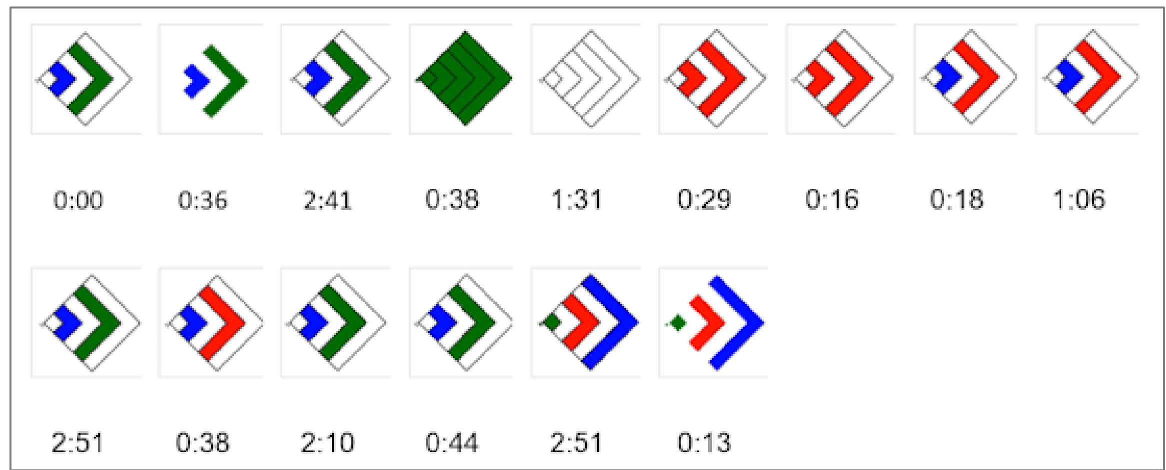


Figure 5.11. George's Week 8 debugging challenge turtle output progression. George changed the modulo operator that controlled the colour selection repeatedly.

```
is_even 5 True
is_even 4 False
is_even 3 True
is_even 2 False
is_even 1 True
```

Figure 5.12. George's Week 8 debug output, printing values for *is_even* function. Debug output shows that the *is_even* function is incorrectly returning True for odd numbers and False for even numbers.

Like George, Mark attempted to use the debug print-statements, but did so in a novel fashion by copying and pasting the included debug print-statement code into the *is_even* function. However, rather than working right away, this introduced a number of compiler errors due to variable names not matching. Although Mark corrected these errors, he did not act on the information provided by the debug print-statements right away, and instead spent considerable time modifying the sequence of the colours in the list provided without success and without using debug print-statements to provide any context. When he did return his focus to his own debug print-statements, he finally realized that the odd and even results were incorrect and stated, "*So, now it's the exact opposite.*" (Week 8: 1:47:37). He then immediately made the correct code change to the *is_even* function to return True when even and False when odd. Mark was the only student to correctly identify that the incorrect code "if n % 2 == 0" was True when n was even and changed it to the correct "if n % 2 == **1**". As a result, Mark's code returned True for even numbers and False for odd numbers, as expected. In total, Mark ran the

program 23 times, introduced and undid seven of his own errors, added 12 debug print-statements to the program, and took the longest of all participants to complete the debugging challenge at 27:40 (see Figure 5.13).

Figure 5.13. Marks' Week 8 debugging challenge turtle output progression.
Mark ran the code repeatedly while modifying only the debug output, not the turtle output.

5.7.2. Weeks 7/8 Student Debugging Challenges

Although the final challenge proved to be too difficult for two of the six students, most of the successful students exhibited challenges as well. These challenges are summarized in Table 5.9 and described in more detail with student examples below.

Table 5.9. Weeks 7/8 student debugging challenges

Challenge	Counterproductive use
Introducing new bugs.	Changing the code unintentionally introduces new bugs
Getting stuck on a repair.	Vacillating when faced with a repair decision so much that they stop, shutdown, or give up.
Making excessive changes.	Repeatedly changing the same line or block of code rather than stepping back to consider alternatives or verifying that the code in question is the correct code to change.
Failing to locate bugs.	Unsystematically searching until all potential causes or candidates have been exhausted, leading to stopping, shutting down, or giving up.
Reading the code sequentially rather than functionally.	Reading the code as text rather than as a sequence of commands with a specific order. Thus, taking additional time and reviewing unnecessary code.
Misidentifying the role and function of debug print-statements.	Students incorrectly assumed that debug print-statements affected the state or flow of the program rather than just reporting on it.

A number of students introduced bugs while debugging the final challenge: Oscar (2 bugs), Brian (4 bugs), Mark (7 bugs), George (9 bugs), and Lucas (15 bugs). Although Oscar only introduced two bugs, he also only tried two changes before undoing them and giving up on the challenge. Brian's bugs were related to incorrectly trying to remove the outside line for the squares, and were all removed. Mark and George both repeatedly changed the modulo operator while trying to figure out the colours, but both ultimately discovered that the modulo operator for the colours was not the cause of the bug and undid their changes. However, Lucas repeatedly changed a single line, ran the program, did not see the correct change, undid the change, and repeated the process without success. Positively, none of the students introduced a new bug that they did not undo.

As previously mentioned in the strategies section, Mark copied some of the provided debug print-statements into his *is_even* function, which introduced a number of compiler errors at the start of his session. He then spent a few minutes attempting to fix these errors and at one point said that he should just undo all of the changes. I suggested that he correct them first, to see what information they provided, and then decide. However, by making this unnecessary change so early in the session, he not only took longer to complete the challenge, but his tone changed to one of frustration early on. Although his tone would improve as he uncovered the clues necessary to

complete the challenge, this unnecessary change early on did result in evident stress and frustration.

Although ultimately successful in debugging the code, George got stuck on how the modulo operator worked. As a result, he made repeated changes to the operator constant to try to fix the colours for the squares, all the while ignoring the output in the debug print-statements that showed the value of the modulo statement. This incorrect fixation on the operator occupied a significant amount of his debugging time, and demonstrated a lack of understanding of the output from the debug print-statements. While running the program, one of the debug print-statements printed the output from the modulo operation but George (and others) ignored or misunderstood the output. This ultimately led to George and others making excessive changes rather than making a change and verifying or testing the output successfully.

Two of the students, Oscar, and Lucas, failed to use the provided debug print-statements effectively, failed to locate the bugs, and ultimately failed to complete the challenge. Oscar spent the first half (4:05) of his session reading the code, first sequentially from top to bottom, and then jumping around randomly. He made a single change that resulted in a single blank diamond, undid that change, immediately made another change that resulted in a compiler error and then gave up, stating that "*I can't do it. I don't know.*" (Week 7: 1:41:36). At this point he was reminded by the researcher of the commented-out debug print-statements on lines 38-41 that he had not tried yet. Immediately, he uncommented each of the provided debug print-statement lines, one at a time, running the program after each change to look at the resulting output. Initially he had an excited tone as he ran the code, only to finish with frustrated statements of, "*It didn't change*." (Week 7:1:43:01) and "*It's the same thing.*" (Week 7:1:43:14). He stopped at 8:04 and exclaimed, "*I still can't do it.*" (Week 7:1:43:44). Oscar's progression is shown in Figure 5.14. Unfortunately, Oscar was unable to locate the bug and misidentified the role and function of the debug print-statements. Apparently, he expected them to repair the code rather than provide information about how the code might be malfunctioning. He ran the program two times, introduced, and undid two bugs, uncommented but did not use the provided debug print-statements, and invested 8:04 working on the problem before giving up.

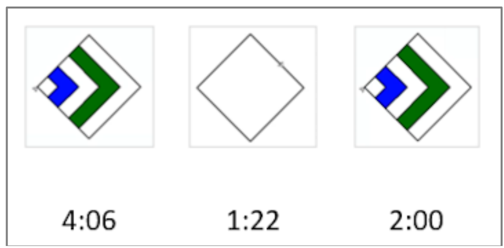


Figure 5.14. Oscar's Week 7 debugging challenge turtle output progression.
Oscar did not successfully complete the challenge.

Lucas approached the final debugging challenge very differently from the other students. He started off by quickly scanning the code, jumped down to the bottom where the debug print-statements were commented out, and uncommented all of them all at once. He then ran his program and exclaimed, "*It does nothing different.*" (Week 7: 1:51:44), referring to the debug print-statements. He then correctly uncommented the code that changed the pen colour to remove the square outlines, ran the program to produce the correct output, but then unexpectedly undid the change and never returned to it. Following these initial changes, he then began changing a variety of lines of code in rapid succession without providing explanations or speaking much at all. After several mistakes, he exclaimed out loud that he was, "*Just messing around.*" (Week 7: 1:53:55). Although some of his mid-challenge changes were focused on the colours, he eventually abandoned those and started changing positioning and angles, all of which he undid. Finally, near the end of his session, sounding very frustrated, he stated, "*I don't know*." (Week 7: 1:59:06), tried a few more changes, and then stopped. In total, he uncommented all of the debug print-statements but ignored their output, ran the program 19 times in 10 minutes, and introduced and undid 15 errors. Lucas' progression is shown in Figure 5.15. Like Oscar, Lucas misidentified the role and function of the debug print-statements and was unable to locate the cause of the error.

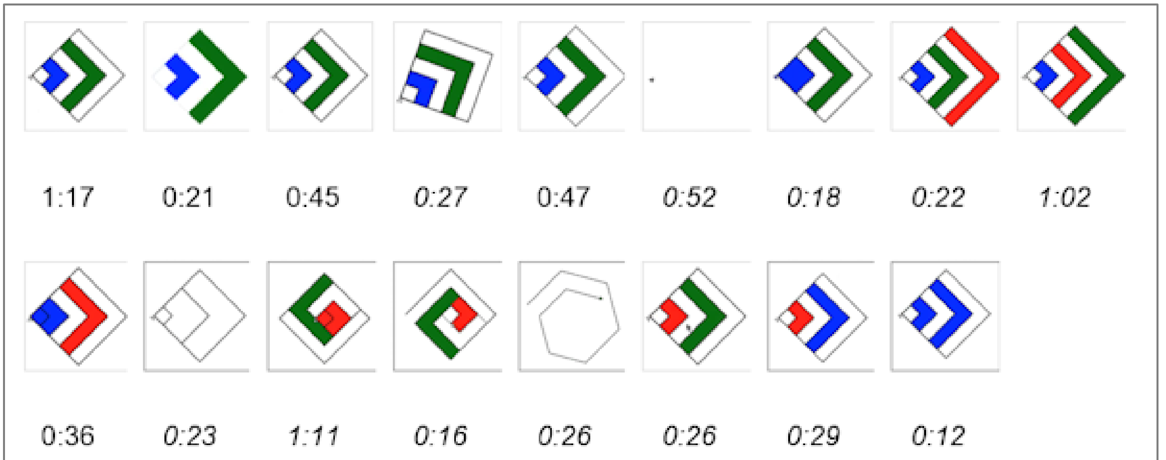


Figure 5.15. Lucas' Week 7 debugging challenge turtle output progression.
Lucas did not successfully complete the challenge.

5.7.3. Weeks 7/8 Debugging Times

Only six of the nine participants took part in the Weeks 7/8 challenge, due to a lack of time. As shown in Figure 5.16, for these six participants the minimum time to complete was 3:20 and the maximum was 27:40 (mean: 12:36, median: 9:46, SD: 8:36). Four of the students successfully completed the challenge, while two did not. Of the six students, two completed the challenge with little to no use of the provided debug print-statements (Peter and Brian), two completed the challenge and used the debug print-statements extensively (George and Mark), and two students failed to complete the challenge and made no effective use of the debug print-statements (Oscar and Lucas).

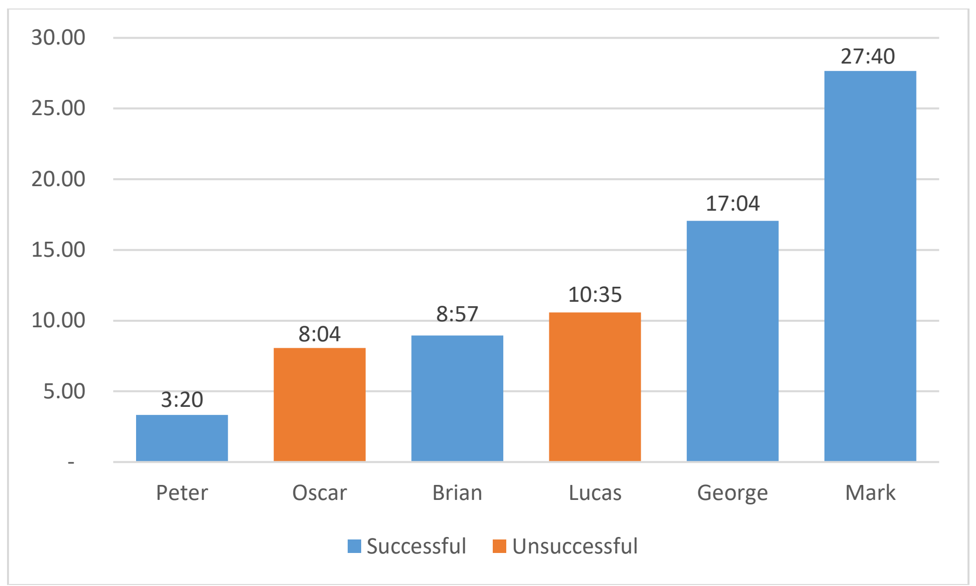


Figure 5.16. Weeks 7/8 debugging challenge times (in minutes, sorted by time).
* Successful means participants successfully debugged the program. Unsuccessful means participants did not complete the debugging challenge and the time shown is when they stopped.

5.7.4. Weeks 7/8 Post Self-Efficacy Ratings

Unlike the previous two challenges, students were only asked to rate their level of debugging confidence after completing the challenge, and unfortunately due to researcher error, the first two students were not asked to provide a rating. As a result, the self-efficacy ratings collected during Week 8 only include four students (Brian, George, Mark, Peter). Peter and Brian maintained similar ratings as in previous weeks, whereas both George and Mark reported lower scores. In the case of George and Mark, both took significantly longer (see Figure 5.16) to complete the challenge than their peers, and both noted challenges trying to use the debug print-statements effectively. However, George and Mark were the only two students to correct the actual problem with the code, whereas Peter and Brian worked around the problem and Lucas and Oscar did not successfully complete the challenge at all. Table 5.10 lists students' self-reported ratings and comments on their ratings.

Table 5.10. Weeks 7/8 student self-reported self-efficacy ratings

Student	Pre-	Post-	Diff.	Student answer to why pre- and post- difference.
Brian	-	4.5	-	*I don't know, just like around the middle, maybe a bit higher.*
Carl	-	-	-	-*
George	-	4.0	-	*I feel like I can debug simple code, but not too complicated and like four is kind of in the middle.*
Jason	-	-	-	-*
Lucas	-	-	-	Not asked**
Mark		3.6***	-	*Because I'm not that good. I don't think I'm using print statements to solve code and so it just takes me a while because I don't know exactly where to put them. And then sometimes I'm not good with like seeing how the output can help too.*
Oscar		-	-	Not asked**
Peter		5.5	-	*At first, I was pretty confused. I didn't know what was going on. But after that, I figured it out pretty quickly, I think.*
Walter	-	-	-	-*

* Three students did not participate in the Week 7/8 challenges due to time constraints.
** Two students during Week 7 were not asked post-challenge to report their score.
*** Mark initially stated his score as 4.1 but while explaining his score reduced it to 3.6.

5.8. Summary of Student Self-Reported Self-Efficacy Ratings

At three points during the study, students were asked to self-report their level of confidence to debug Python code, on a scale from one to seven. Figure 5.17 shows the individual scores for all students over the span of the course. Four different patterns are described: a drop in confidence following a difficult debugging session (Carl), two over-confident reports (Lucas & Oscar), a downward trend (Mark), and two students whose scores stayed the same throughout (Brian & Peter).

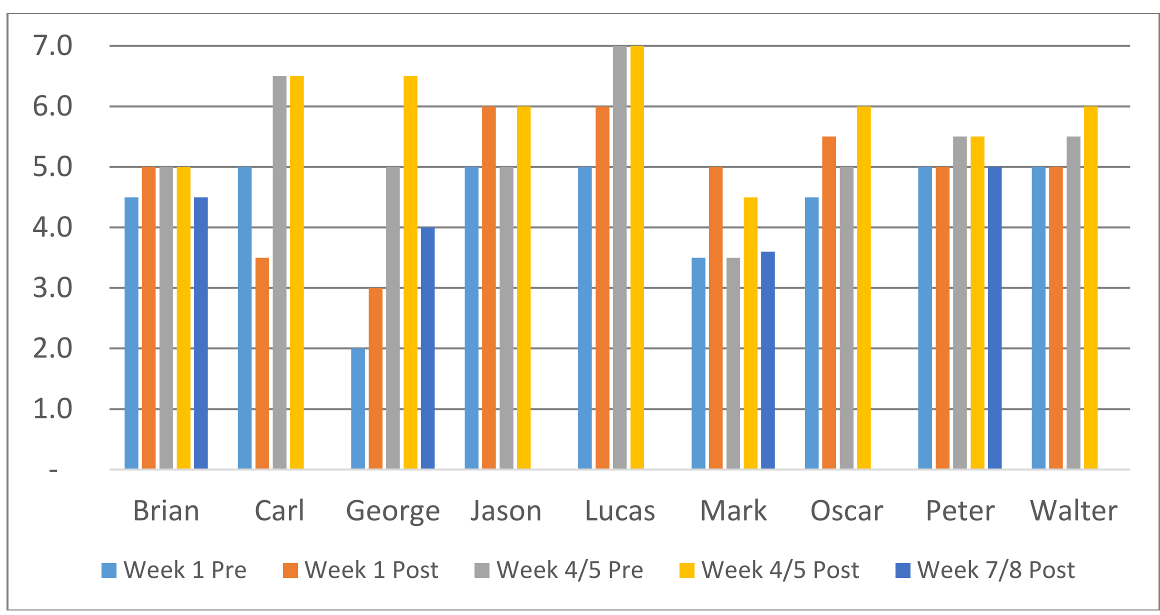


Figure 5.17. Student self-reported debugging confidence scores.

As shown in Figure 5.17, Carl reported the only decrease in confidence (5.0, 3.5) following his struggling with the final challenge during Week 1, but then reported increased confidence (6.5, 6.5) during Weeks 4/5 but offered no conclusive reason for the increase.

On the other hand, Lucas and Oscar never decreased their reported scores and instead reported ever increasing scores over the first two challenges. As well, both asked if they could choose a score higher than seven, with Lucas even stating that they thought they were really a nine (out of seven) at the end of Weeks 4/5. They have been grouped here because they both reported struggling during their sessions but then reported higher scores as a result of completing the challenges. When asked why they increased their score, Oscar noted, "*Because I honestly didn't think I was going to do that. I kind of like, I don't know, boosted my confidence.*" (Week 4: 1:49:48), and Lucas noted, "*When I started noticing more stuff, I got more confident.*" (Week 4: 2:08:56). Unfortunately, due to a researcher error, neither Lucas' nor Oscar' final scores were captured during Week 7. This is unfortunate because both students failed to complete the challenge, and both had sounded downtrodden at the end of their final challenge session.

In contrast, Mark reported a higher level of confidence at the end of each session, but his week-to-week scores exhibited a downward trend over the weeks. He

started with 5.0 during Week 1, then 4.5 for Week 4, and finally a score of 3.6 for Week 8. During Week 8, Mark initially stated a score of 4.1, "*I feel like a 4.1*." (Week 8: 1:48:48), going on to clarify, "*Because I don't feel like I'm a four, but I don't feel like I'm any higher than a 4.5.*" (Week 8: 1:48:56). However, while explaining his reasoning for his initial 4.1 score, he noted that he was not that good at using the debug print-statements demonstrated in the class and then suddenly decreased his rating from 4.1 to 3.6 (a 0.1 increase over half of seven), noting, "*So, that's why I thought actually maybe a 3.6 then.*" (Week 8: 1:49:29). In discussing it further to clarify the lower final confidence score, Mark noted that he had taken longer than expected to complete the challenge and had trouble with the debug print-statements, which affected his confidence.

Finally, Brian and Peter both consistently reported the same scores (between 4.5 and 5.5) throughout their three sessions and either did not change their scores or only changed them marginally (0.5) from the start of the session to the end. When asked why there was no change, Peter replied, "*I think it was a pretty accurate guess.*" (Week 1: 2:34:41), and Brian replied, "*It didn't feel like it changed.*" (Week 5: 1:44:59).

5.9. Student Descriptions of Debugging

Over the course of the eight-week study, students were asked three times to provide a description of the term debugging. The question was posed as, "In a sentence or two, how would you describe debugging to a friend?" The answers were scored according to whether students identified the three central tenets of debugging described and discussed in the course: identifying that a bug exists ("identify"), locating the bug ("locate"), and fixing the bug once located ("repair").

During Week 1, students were asked to describe debugging before any instruction on debugging. None (0/9) of the students mentioned "identifying" that a bug existed, only two (2/9) noted "locating", while most of the students (8/9) included the term "fixing" in their descriptions. During Week 2, participants were again asked to describe debugging after they had been exposed to three different debugging situations: 1) they had completed their first debugging challenge the previous week, 2) they had submitted their own code with bugs in it during this session, and 3) they had watched the researcher debug their code live in class. Following these debugging experiences, two

(2/9) noted “identifying”, three (3/9) included “locating”, and all of the students (9/9) included the term “fixing” in their descriptions. Finally, during Week 6, after a class on an explicit debugging process, watching the researcher demonstrate debugging, and having completed two one-on-one debugging sessions, participants were once again asked to describe debugging. The results were mostly unchanged from Week 2, two (2/9) noted “identifying”, four (4/9) included “locating”, and most (8/9) included “fixing”.

Although most students’ descriptions were largely unchanged week to week, one student, George, demonstrated a clear progression over the course of the study. First, his Week 1 response identified only fixing: “*Debugging is fixing broken code so that it works.*” However, at the end of Week 2, his definition expanded to include both identifying and fixing: “*Debugging is recognizing that there is an error in the code and fixing it.*” Finally, during Week 6, his definition expanded to include all three elements: “*Identifying a problem in the code, finding it, (and) then solving the problem.*” In addition, although not captured in his descriptions of debugging, at the end of his Week 5 one-on-one debugging challenge, he asked about the green bug icon on the PyCharm toolbar (the debugger button), how the debugger worked, and if the researcher could show him how it worked. After a short demonstration, he was encouraged to play with it on his own and ask any questions.

All student descriptions of debugging are shown in Table 5.11.

Table 5.11. Student descriptions of debugging (weeks 1, 2 & 6)

Student	Week 1	Week 2	Week 6
Brian	*Debugging is where you fix errors that are within a program, I think.*	*Debugging is fixing errors or malfunctions in a program, that's basically it.*	*Debugging is fixing an error in program, there are three types of errors: syntax errors, runtime errors, and logic errors.*
Carl	*It just means fixing a line or a part of a code.*	*Debugging is about finding and fixing a problem in your code*	*Debugging is like fixing code and like other things you learn there are strategies to it.*
George	*Debugging is fixing broken code so that it works.*	*Debugging is recognizing that there is an error in the code and fixing it.*	*Identifying a problem in the code, finding it, then solving the problem.*
Jason	*When there is something in the code that's wrong and you have to take it out like e.g., pulling a piece of fluff out of a toy with things in it*	*Debugging is like when there is a problem, and you have to undo it, but you have to write it on a computer or it's like taking something out of a toy that's not supposed to be there*	*It's like taking a needle out of a haystack except t's not impossible to find. You have to dig around the hay or the code and find the needle or the problem. Then, you pull out the needle, or fix the code.*
Lucas	*I would describe it like you fixing a car or fixing toy so what you do in coding is debug the code of any bugs*	*Debugging is like making spelling mistakes and needing to erase it and put the correct thing in*	*Like if you make mistake when drawing or writing you erase it and fix it*
Mark	*I would describe debugging as fixing an error in a program.*	*Debugging is finding an error in a code, and properly fixing it. I think if you don't really know how to code. you can't exactly fix or find errors, so it's hard.*	*I would describe debugging as locating and fixing an error in coding. Debugging could also be described as fixing a problem in coding with simple methods.*
Oscar	*Fixing errors in the code; changing something in the code that you don't want to be there*	*Debugging is fixing errors in a code*	*Debugging is fixing mistakes and changing lines of letters and numbers*
Peter	*Debugging is checking code for errors and then fixing them*	*Finding and fixing problems in code*	*Searching for and fixing broken parts of code*
Walter	*Debugging is like a puzzle. You have to find the thing you did wrong which might be hard a first but once you get the hang of it, it gets easy.*	*Debugging is where u kinda fix the code if there was an error; also, it's kinda hard at first (be)cause you don't really know how to do it; (be)cause you're just starting.*	*Debugging is kinda hard if u don't know much about coding but if u do it's kinda like writing on a word doc(ument).*

Chapter 6.

Discussion

This chapter discusses the findings of the study in relation to the literature on novice-programmers' debugging. It is divided into seven parts: (1) a review of the research aims and research questions; (2) a summary of the key findings; (3) an interpretation of the findings in relation to prior research; (4) a discussion of the implications of the findings; (5) a discussion of the limitations of the study; (6) a set of recommendations based on the study; and (7) a concluding summary.

6.1. Research Aims & Research Questions

This research aimed to identify the debugging strategies and challenges that upper elementary-school aged novice programmers demonstrate before explicit debugging instruction and then the observed effects and their responses to an explicit debugging strategy taught within the context of a Python programming class. Four research questions were pursued:

RQ1: What strategies do novice elementary school students employ when debugging Python code?

RQ2: What challenges do novice students face when debugging?

RQ3: What effects does an explicit debugging strategy have on student debugging strategies?

RQ4: In what ways do students respond to explicit debugging instruction?

6.2. Summary of Key Findings

Before explicit instruction on debugging, students demonstrated a number of common strategies and challenges. For syntax and runtime errors, students relied on external information from the editor and compiler to identify, locate, and repair bugs. For semantic errors, students used a combination of prior programming knowledge, previous

debugging experience, and reading and tracing the code to identify and locate bugs. Once located, students primarily used prior knowledge, comparison to working code, and guessing-and-checking to repair bugs.

In addition to common strategies, students also exhibited common challenges that slowed them down, misled them, or prevented them from debugging successfully. The most common challenges observed were students introducing new bugs, making excessive changes to code, reading code sequentially rather than functionally, and applying fixes that worked around rather than repaired the underlying bugs.

Following explicit debugging instruction during the course, two changes were observed in students' debugging strategies. First, students began running their programs as their first action, which both generated compiler or runtime errors to provide clues and also allowed them to observe the turtle output to help them understand the control flow of the program. The second change observed was that some of the students began utilizing debug print-statements to identify and locate bugs; including some creating their own print-statements.

Finally, although some students showed signs of adopting the explicit debugging strategies, the results were mixed, with three main responses identified. One group of students adopted the explicit debugging strategies taught in the course and used them successfully, but exhibited signs of excessive cognitive load while doing so. A second group, although successful, either did not use the strategies taught or did so sparingly, and instead primarily relied on their own strategies. A third group did not use the taught strategies, did not appear to understand their role, and ultimately gave up during the final challenge. The findings of this study suggest that students who adopted the explicit debugging strategies were successful, but appear to require additional practice to master them. Understanding the students who did not use the taught strategies will require further study.

6.3. Discussion of Findings

This section will discuss the findings as they relate to the research questions.

6.3.1. Student Debugging Strategies

This section will discuss the key debugging strategies that students demonstrated as they attempted to complete the debugging challenges before receiving instruction on explicit debugging strategies.

At the start of the study (Week 1), students were initially introduced to syntax errors during the first three challenges. Although it is critically important to be able to identify syntax errors and read and react appropriately to their resulting runtime errors, they are often overlooked or characterized in the research literature as “easy errors”. Although commonly counted during studies looking at error frequencies (Ahmadzadeh et al., 2005; Ripley & Druseikis, 1978; Spohrer & Soloway, 1986), research into how to handle them has often been ignored, due to their low frequency of occurrence (Boies & Gould, 1974) and the ease with which they are identified and corrected even by novice programmers (Youngs, 1974). However, for both “true novices” (Kessler & Anderson, 1986) and younger novices (as in the present study), strategies for handling syntax errors are critical because they are the first debugging barrier that novice programmers face when learning to program. Failure to master effective approaches to them could result in a barrier to more complex programs and result in an inability to move on as a programmer.

One of the first strategies observed by students when faced with syntax errors was to use hints provided by their code editor (PyCharm). Using code editor hints was not discussed in early studies because early editors were relatively primitive, and computers did not possess the spare processing power necessary to scan for and alert the programmer of errors while they were working with the code. As a result, during early studies programmers relied on the compiler to report syntax errors when the programmer stopped writing and submitted their program for inspection via compilation or interpretation. This has changed as computers have become more powerful, and as a result, modern studies have mentioned the use of tools and online resources (Fitzgerald et al., 2008; Murphy et al., 2008). The students in this study, having used the PyCharm editor before, demonstrated that they were already familiar with using it to locate and identify syntax errors. As a result, they were aware that code that was underlined with a red line contained a syntactical error, and so they used these hints to home in on and correct such errors quickly. For example, during Week 1, after pasting

the bugged code into the editor, most of the students immediately moved their mouse to the first line that had a red underline, read the underlined text, and immediately corrected the underlined error.

Like editor hints, students in this study had also already been exposed to compiler error messages when their programs "stopped working" or "crashed." As a result, when they encountered such errors after running their programs, they exhibited two strategies: immediately moving their attention down to the output window of the editor to read the error message, or stopping and asking for help from the teacher. Most of the students in this study reacted to compiler errors by immediately shifting their focus to the output window to read the error message, using the information to locate the error, and then correcting it. For those who were less familiar or unsure when an error occurred, typically a prompt from the researcher to read the error message was sufficient for them to locate the line and the incorrect code as well. However, a few students, even after being reminded to read the error messages, still stopped, and asked for help rather than attempting to correct the error on their own, indicating a misunderstanding or a learned helplessness whereby their initial reaction to an error is to ask for help rather than try to fix the error on their own. Perkins et al. (1986) attribute this to the students not wanting to be seen making mistakes, or what Dweck & Elliott (1983) characterize as performance-oriented motivation.

As the debugging challenges increased in difficulty from the simpler syntax and runtime-related errors to the more challenging semantic errors, students' strategies changed as well. This was especially noticeable during Weeks 4/5 and 7/8, because the number of lines of code exceeded a single screen and so students had to scroll their editor window up and down to see all sections of the code. As a result, students started off looking for editor hints like they had during Week 1 by scrolling the editor screen up and down. When that failed to identify any obvious (syntax) errors, they switched to reading the code line-by-line, which resulted in two observations: code comparison and guessing-and-checking.

During Weeks 4/5, after reading the code line-by-line, most students noticed that the blocks of code for the three squares were similar to each other, and thus they could identify bugs by comparing the working code to non-working code. This most closely aligns to "pattern matching" from Murphy et al. (2008), although Murphy et al.'s

examples were more focused on code example found via web searches rather than the more general comparison to known working code. In this study, for those students who compared the code to similar blocks of code, they began by comparing similar code, looked for and changed what was different, and then ran the code to see if their change was correct. In most cases this was an effective strategy, as they used the known-good code and their prior knowledge or experience successfully to identify what was wrong and make the correct changes.

For those students who did not recognize the similar code, they instead started off by reading the code, suggesting that they were tracing the code line-by-line and mentally simulating each command. During Weeks 4/5 in particular, this line-by-line code tracing approach led to them narrowing down the general area of the bug, but it did not always result in identifying a specific line of code that needed to be changed. As a result, students were often left unsure of what needed to be done to fix the bug and what line(s) to change. When this happened, students would often make a change to the code (guess) and then run the program to see if they were correct or not (check). This guessing-and-checking was often successful, demonstrating that students had a weak understanding of possible solutions, hypothesized possible repairs, and then verified their changes. However, as discussed later in the challenges section, when guessing-and-checking did not work, it resulted in excessive counterproductive changes that frustrated students and caused them to become stuck or give up.

Overall, before explicit instruction in debugging strategies, students exhibited some previously learned behaviours and general debugging strategies. To start, their experience with both the PyCharm editor and other text-editing tools helped them identify syntax errors using a familiar red underline to signal an error and combined with context clues from the surrounding code or their own knowledge of command syntax, they easily corrected most of the syntax errors. As the challenges shifted towards semantic errors, the students exhibited the ability to spot differences through pattern matching or guessing-and-checking based on their understanding of the code or prior programming experience. In most cases, for the first set of challenges, the existing strategies that students had, combined with their programming knowledge allowed them to identify, locate, and repair all of the challenges successfully. However, as will be discussed in more detail in the next section, when their existing strategies were insufficient, students would run out of options and get stuck or give up.

6.3.2. Student Debugging Challenges

In addition to demonstrating a variety of strategies during their debugging sessions, students also exhibited a number of challenges that slowed them down, led them astray, or resulted in unproductive or counterproductive actions. This section will discuss the key challenges observed as students attempted to complete the debugging challenges.

Even when students successfully debugged the challenges, they often introduced new bugs while attempting to repair the existing ones. This is a common challenge noted early on by Heilman & Ashby (1971), who observed that any changes to the source code, even the addition of debug print-statements, risks introducing more bugs, as was observed during Week 8 (Mark). Bonar (1985) suggests that students introduce bugs because they encounter a gap or inconsistency in their programming knowledge which results in an impasse. This in turn leads to them guessing at solutions, which is prone to introduce new bugs. Perkins et al. (1986) suggested this occurs because novices act unsystematically, tinkering rather than debugging systematically. Nanja & Cook (1987) characterized novices by their introduction of new errors into the code, as compared to experts who did not, and intermediates who did so only occasionally. Fortunately, the students in this study undid all of the bugs they introduced, showing that they were aware of the changes they had made, kept track of those changes, and undid them when they determined that they did not work.

As mentioned in the previous section on debugging strategies, some students made an excessive number of changes while debugging, especially when guessing-and-checking. Perkins et al. (1986) described students who moved too fast, and tried repairing their code in rapid-fire succession, without reflection or apparent forethought, as “extreme movers” (p. 42). This pattern of excessive changes was most apparent for two cases during the study: Oscar during Week 4 and Lucas during Week 7. In Oscar’s case, he made seven incorrect changes (compared to an average of two by the other students) but as noted above, he undid each one and was ultimately successful in completing the challenge. His approach was unsystematic, and he appeared to stumble upon the solution rather than arrive at it logically like the others. This led to him noting, “*I honestly didn't think I wasn't going to do that.”* (Week 4: 1:49:48). During Week 7, Lucas

exhibited a similar rapid-fire succession of changes, but unlike Oscar during Week 4, Lucas did not stumble across the solution and ultimately gave up on the challenge.

Perkins et al. (1986) suggest that students adopt an “extreme mover” behaviour to appear busy and to avoid failure by persisting. However, they also note that this behaviour can lead to loss of self-confidence and discouragement to continue programming. In Oscar’s case, he acted confidently as he made his mistakes during Week 4; but during Week 7, he lost all confidence when his actions did not work as expected. In Lucas’ case, he seemed to fit Perkins et al.’s characterization of looking busy and avoiding failure when he stated that he was, “*Just messing around.*” (Week 7: 1:53:55). Upon reflection, this behaviour of acting busy to avoid failure aligns with the students being performance-oriented rather than learning-oriented (Dweck & Elliott, 1983), something Papert (1980) had noted separately as well. Although throughout this course bugs were characterized as “no big deal” (NBD) and something that happens all of the time (including while the teacher was teaching), some of the students appeared unwilling to be wrong, and as such persisted unproductively to appear busy and hope to luck into a solution rather than taking a step back, re-checking their assumptions, or looking for clues as suggested by McCartney et al. (2007).

Although the goal of debugging is to identify, locate, and repair bugs, some repairs are better than others. For example, a repair that fixes a root problem in a function called from multiple places within the application can result in a single repair fixing multiple bugs. However, if the root problem is not repaired and instead each symptom of the bug is made to work with the still-buggy code, then the programmer has failed to identify and fix the root cause. This avoidance can result in the modifying of code that did not need to be modified. Further, when a bug impacts multiple areas of the code, as in our example above, it can lead to code maintenance issues because each of the worked-around or avoided pieces of code themselves needs to be maintained according to the still-buggy code. As a result, it is imperative that students identify the root cause of the bug rather than just the symptoms of the bug.

Murphy et al. (2008) called this “working around the problem” and described it as happening because students did not understand what the code was doing and so they replaced it with code that they did understand (p. 166). This approach was exhibited by Mark and Lucas during Weeks 4/5 when they added unnecessary code after or within

the for-loop for square three that was missing its fourth side. The bug in the code was that the range for the for-loop was too small, something that was shown by the debug print-statement outputs as well as the turtle output. In Mark and Lucas' case, rather than identifying and fixing the bug in the range, they both added unnecessary extra turtle movement commands after the loop to complete the square. Although their solution produced the correct output, if they had then been asked to modify the code to draw a different polygon with more sides, their code would not have worked, and they would have had to add to or modify their unnecessary repair code to compensate while the other students could have just changed the single range value.

The last common challenge that students exhibited was reading the code sequentially from top to bottom, like prose, rather than as a series of ordered instructions (Jeffries, 1982; Nanja & Cook, 1987). This behaviour was sufficient during the first two challenges because the code itself was a series of sequential instructions. Therefore, reading it sequentially matched how the computer read it. However, the debugging challenge code for Weeks 7/8 used two functions, one for drawing the squares (*square*) and the other for determining whether a number was even or not (*is_even*). As a result, the order of execution for the final challenge was not sequential and the main bug was contained within the *is_even* function. This resulted in student times being exaggerated the more time they took trying to read the code. As well, students exhibited signs of excessive cognitive load as they tried to both simulate and keep track of the program state as they read through the code sequentially. This is in contrast to expert programmers who maintain only the necessary current state as they traverse the code functionally or who offload the memory work to external sources or mechanisms (Fitzgerald et al., 2008).

6.3.3. Effects of Explicit Debugging Instruction

At multiple points during the programming course, students were exposed to a variety of implicit and explicit debugging instruction. The implicit instruction included organizing the code to make it easier to read, and running the code first and often while debugging to verify changes as soon as possible. The explicit instruction included demonstrating an expert debugging process using a think-aloud verbal protocol, introducing the Wolf Fence algorithm for locating bugs, and introducing and using debug print-statements to trace program flow control and program state. Following these

debugging demonstrations and instructions during the course, the students adopted three new strategies: (1) running the program as the first step in debugging, (2) applying the Wolf Fence algorithm, and (3) using debug print-statements.

One of the first changes observed in students' approaches to debugging after explicit instruction was their running of the program as their first step in debugging. Prior to this (during the first week for example), students focused on finding and fixing typographical errors using the editor hints. As a result, they would read the code first and even attempt fixes before running their programs. However, during Week 2, while debugging the students' code live, the researcher always stated that he was first going to run the student programs first to see what information he could glean in terms of compiler error messages or output variations. When asked what differences between their own approaches and the researcher's approach, three students noted that the researcher ran the program after each fix, rather than as a first step. Following this live demo where students repeatedly saw the researcher running their programs first and after each change, student behaviour changed to running their programs first and often as well. For example, during Weeks 4/5 only Jason and Peter read the code first, while the other seven ran their programs first. As well, during Weeks 7/8, only Peter read the code first, while the other five ran their programs first.

Carver's (1986) five-point debugging process included running the code as the first step. Her rationale for this was that bugs are typically not noticed until the program is compiled and run, so telling the students to run their program at the start and after each change is beneficial. Although this was not explicitly stated during my debugging instruction, most students adopted this strategy because it allowed them to see any compiler error messages, see the turtle output, and also observe the sequence of turtle actions as the program executed. As a result, running the code gave them a sense of what the program did before they tried to debug it as well as after each change.

As the course progressed, students were introduced to two explicit debugging techniques: the Wolf Fence algorithm (Gauss, 1982) for sub-dividing the code to aid in locating bugs, and debug print-statements for tracing program flow and state changes over time.

Although students were explicitly taught how to use the Wolf Fence algorithm as part of a code-along debugging exercise during Week 6, only a single student (George) attempted to use it during Week 8. In George's case, he added a single *print("potato")* line as a marker in his code (as prescribed by the algorithm), then ran his program, read the debug output, and stated that it did not help him. Unfortunately, he placed the print statement at the start of the program, and failed to add the necessary additional statements or move the print around in the code as he worked. It is clear from this single use that George tried to use the algorithm but that he did not recall or understand the need to move the marker or to add more markers to further sub-divide the code. Böttcher et al. (2016) described a similar experience during their study when they tried to introduce the Wolf Fence algorithm, noting that students did not use it and instead returned to "poking around." Unfortunately, given that only a single student tried to use the approach, it is not clear if the lack of use was because of the simplicity of the debugging challenges, and thus this approach was unnecessary, or that the approach was too complicated and students either failed to understand it or chose to stay with approaches they were more comfortable with.

In contrast to the single exercise and demonstration of the Wolf Fence algorithm, print-statements were used throughout the course. To start, print-statements were introduced during the first week as the sole means of program output to familiarize students with how they worked and to implicitly demonstrate their ability to output program state information as it changed. During Weeks 4/5, the challenge code included researcher-provided debug print-statements for each of the three squares to both identify the start of each block of code for each square, and also to output the turtle's state as each square was drawn. Most of the students used the identification print-statements to locate the code for each square but only a single student (Walter) mentioned and then demonstrated reading the output to identify and then locate one of the bugs. The nature of the bugs for Weeks 4/5 did not specifically require using the debug output, but using it made finding the rotation bug much easier. However, even though Walter correctly identified the reason for the bug, he did not use this information correctly. Instead, rather than returning to the debug print-statement line to see what variable was wrong, he mistakenly deleted the first minus sign (-) he found in the code. So, although the debug output helped him identify the issue, he did not use it fully to locate or repair the bug itself.

In the final challenge, a set of commented out (disabled) debug print-statements were provided in the bugged code. Thus, by default, they generated no output. A comment in the code identified them as "some debugging print statements" and students were told that the statements were there if they wanted to use them, but that they were not required to. Of the six students who participated in the final challenge, one did not use the debug print-statements at all (Peter) while the other five uncommented all of them. Brian only looked at the output once at the start, and then proceeded to debug the code without using them. Lucas and Oscar also uncommented all of the debug print-statements, and both noted that they did not change the turtle output and so dismissed them. Both Lucas and Oscar failed to complete the challenge. In contrast, Mark and George both used the debug print-statements successfully. However, their use was also unexpected and is worth discussing in more detail.

In addition to using debug print-statements to locate the bug, George successfully used the output from the provided debug print-statements to identify that the *is_even* function was returning the wrong result. Through careful and repeated reading of the output, he eventually noted that the output was wrong, and that led him to correctly identify that the function was incorrect and to make the correct change. This shows that given the necessary clues (in this case that the function was not working as required), George was able to then locate and correct the error (unlike Walter in the previous example). However, George did not initiate the creation of the debug print-statement on his own and thus without the provided debug print-statement, he may not have found the bug on his own. This finding aligns with the four debugging heuristics provided by Fitzgerald et al. (2008, p. 114) which indicates that students need to be taught to validate their assumptions using (debug) print-statements (or other means).

Like George, Mark successfully used debug print-statements during the final challenge. However, Mark used them differently. Mark started off by copying the provided commented-out debug print-statements from the bottom of the code up into the *is_even* function at the top of the code. He placed these copied lines into the conditional clause where the function returned True and uncommented them. Unfortunately, by doing this, he introduced a number of compiler errors and initially stated that he was just going to undo the change; but the researcher encouraged him to fix the compiler errors first and then decide. After fixing them, Mark reviewed the output but was not initially certain of what it meant. After uncommenting and reviewing the rest of the debug

output, he returned to his own debug print-statements and added a copy of them for when the function returned False. It was during this interaction and subsequent inspection of the output that he realized that the function was returning the wrong values. This led him to correct the bug in the function so that it returned the correct values.

Although George made a similar discovery, Mark, through his detailed debug print-statements, was the only student to identify and repair the single change that fixed the function. This deep dive into his own debug print-statements and then the encouragement to follow that ultimately correct pathway suggests that additional practice and support is needed for students to use print-statements early on, when their code comprehension level is still developing, to validate and verify their code. In the case of the final challenge, the modulo operator was used, which was unfamiliar to the student. Although the necessary debug output was available for them to see both how the modulo operator worked and also the buggy *is_even* function, most students failed to follow the steps necessary to identify, locate, and repair the bug efficiently. Fitzgerald et al. (2008), when discussing debugging heuristics (Figure 2.7), noted that students need to be made aware of their limited ability to keep the program in their heads along with the program state, and as such need to be encouraged to offload this memory load such that they can inspect and verify so they can identify, locate, and repair bugs effectively.

6.3.4. Student Responses To Explicit Debugging

Unfortunately, few studies have examined student responses to explicit debugging instruction. Of those that have, Murphy et al. (2008) noted that (university computer science) students were positive towards undertaking debugging activities while learning programming, with one stating, "*All students should do this [debugging exercises] – it is really good for you*." (p. 167). In a study similar to the present one, Ko et al. (2019) noted that 73% of their high-school-aged participants reported that they understood the debugging strategies, but only 53% reported that they were helpful. Ko et al. noted that rather than use the systematic strategies they had been taught, students engaged in rapid cycles of shallow editing and testing, similar to the previously discussed guessing-and-checking from this study. This appears to align with Lowe's (2019) comments that the newly taught strategies still require the slower System 2, and so novices stick to what they have already automated (System 1) because it is faster and is something they already know.

In this study, student responses to the explicit debugging instruction were observed during the final challenge, and three general responses were identified. One group of students used the most strategies to successfully complete the challenge, but needed assistance to do so and showed signs of excessive cognitive load. A second group made no use of the taught strategies, and instead used their own strategies successfully. The third group made no use of the taught strategies, did not appear to understand their role or how they worked, and did not successfully complete the challenge.

The first group consisted of George and Mark, who successfully applied the taught strategies with some instructor support. George attempted to use both the Wolf Fence algorithm and debug print-statements during the final challenge. However, as previously noted, his Wolf Fence attempt consisted of a single *print("potato")* line at the top of his program that he did not return to or use after running the code one time. In contrast, he used the debug print-statements effectively to both identify that the *is_even* function was returning opposite values to those intended, and how to fix it. His repair fixed the buggy *is_even* function and required only two code changes. As a result, he showed a clear understanding of the value of the debug print-statements and how to use them effectively to debug.

Similarly, Mark used the debug print-statements to determine that the *is_even* function was incorrect, locate the incorrect line, and repair it. However, Mark struggled with the debug print-statements initially because he tried to copy-and-paste the provided lines into the function directly, which resulted in compiler errors. He wondered aloud if he should undo this exploration, but I suggested he correct the errors first and then decide. Fortunately, after making the necessary corrections, he was able to use the information from his custom debug print-statements to correctly identify the issue and correct the single buggy line. He was the only student to make the single correct change.

The second group consisted of Peter and Brian, who both completed the challenge quickly and with little to no use of the explicit debugging techniques taught in the class. In Brian's case, he uncommented the provided debug print-statements, ran the program, looked at the output, and then ignored it while he traced the code instead. Peter did not uncomment the debug print-statements and instead traced the code

closely, ran it a few times, and noted that the order of the colours was incorrect. Both students changed the if-statement for the colouring rather than repairing the *is_even* function. As a result, they had to move two lines around to work *around* the incorrect *is_even* function. Although they did not use the strategies taught in the course, their times to complete the challenge were among the lowest, and their approaches the most systematic.

Unlike the first two groups, the third group, which consisting of Lucas and Oscar, did not use any of the explicit debugging strategies taught in the course. They appeared to misunderstand what the debug print-statements did, and as a result failed to complete the challenge. In Lucas' case, he uncommented all of the commented-out code at the bottom of the code, including the debug print-statements. However, upon running the program, he stated that it just output a bunch of text but did not fix the turtle output, and so he ignored the output of the debug print-statements thereafter. Instead, he tried a number of seemingly random changes in an ultimately futile attempt to stumble across something that would provide a clue to the underlying bug. This haphazard approach clearly demonstrated a lack of understanding of both the explicit strategies and how the program worked. As a result, Lucas made an excessive number of edits, never made any progress, and ultimately gave up.

Oscar's actions also showed a lack of understanding of the debug print-statements. He read the code aloud slowly from top to bottom, then jumped back to the top and read the code again. However, rather than tracing the code (and demonstrating an understanding of the flow), he read it sequentially as though it was prose. He eventually settled on a suspect area of the code, made a single change, got a blank result, and stated that he did not know what to do. When reminded of the debug print-statements, he excitedly uncommented them line-by-line, running the program after uncommenting each line. However, after running the last line and seeing no change in the turtle output, he remarked that the debug print-statements did not do anything, and gave up. He was clearly lost, did not understand what the code was doing, and did not formulate a plan to debug the program.

The first group of students represents the closest to expected results, because they not only tried to use the explicit strategies but in doing so, they found and fixed the correct bug. This aligns with the outcomes expected based on the recommendations

from Fitzgerald et al. (2008). However, they also struggled unexpectedly, and required support from the researcher to use the debugging strategies successfully. In addition, they both reported a lower self-efficacy rating at the end of the challenge, even though they had used the strategies successfully.

The second group was successful using their own strategies, rather than the strategies taught in the course. I suspect that the bugs did not present enough of a challenge to require them to need the explicit strategies, so they were able to reason their way through the two bugs and solve the problem. A more challenging exercise, or one that required explicit tracking of program state, may have been necessary to require them to try the debug print-statements or the Wolf Fence algorithm. Unlike the first group, these students reported the same self-efficacy ratings throughout the three challenges, and at the end they maintained that nothing had really changed with regard to their confidence in their debugging abilities.

The third group was a surprise, but was also the most similar to what Ko et al. (2019) noted, namely that even with explicit instruction, these students either did not learn the strategies or chose to revert back to their own less-effective approaches when challenged. Ironically, these two students reported the highest self-efficacy ratings during the first two challenges but unfortunately due to a procedural error, their final confidence ratings were not captured. However, they were both downtrodden at the end of the final challenge when they failed to solve it.

Upon reflection, it is clear that additional instruction, practice, and assessment are needed to support all three groups. The first group needed to build up their confidence, and to do that they appear to have needed additional practice using the debug print-statements. The second group, although successful with the challenge, may also have benefited from practice so they could see the benefit of at least using the output from the provided debug print-statements. Finally, the third group would likely have benefitted from being identified earlier in the course, so that I could have investigated and determined why they were not apparently learning how to use the debug print-statements. Although both students in the third group reported feeling confident, and were successful during the first two challenges, upon closer inspection they did exhibit excessive mover tendencies (Perkins et al., 1986) and excessive errors that may have indicated they were having trouble earlier on.

Before closing out this section on student responses, I want to draw attention to one of the students, George, who demonstrated a clear interest in debugging throughout the course. Following his Week 5 debugging challenge, he asked what the green debug icon (pictured as a green bug) on the PyCharm toolbar was used for. This led to a demonstration and discussion of the Python debugger available within PyCharm, the use of breakpoints, the watch window, and the debugger's ability to step through code line-by-line. George was the only student to ask about this and was encouraged to try it out following the demo; but no further mention of it was made and no further questions were asked. George was also the only student to attempt to use a debug print-statement to split the code, as demonstrated in the Wolf Fence algorithm, and he made use of the debug print-statements to identify, locate, and repair his bug. Finally, his definition of debugging evolved throughout the course, and he was the only one to include all three elements (identify, location, and repair) for his final definition. Looking back, it is clear that George was interested in debugging and that may have played a part in his willingness to try all of the debugging strategies, even if not always successfully.

6.4. Practical implications

This section will discuss the most significant findings and their implications.

First, as noted by Ko et al. (2019), unless student feel confident that they can perform the explicit debugging strategies on their own, they may be reluctant to use them, especially while being observed, even when their use has direct benefit to them. This lack of confidence implies that additional support and practice may be necessary to increase their confidence and work towards automating these processes, as suggested by Lowe (2019). During this study, the actions of Mark and George during the final challenge, demonstrated that students are capable, for example, of using debug print-statements successfully to identify and correct the errors, but that they required additional support from the researcher to do so. Although these were the only two students to use the approach successfully (with support), this finding suggests that with additional support more students could potentially integrate these strategies as well.

In line with lack of confidence using the strategies, some students appear to have adopted a performance-oriented mindset when debugging rather than a learning-oriented mindset (Dweck & Elliott, 1983). As a result, while debugging, they appeared

unwilling to fail or “look bad” in front of the researcher and as a result made themselves appear busy and productive rather than ask for help or make a mistake (Perkins et al., 1986). This manifested as “excessive moving” (Perkins et al., 1986), which resulted in them either randomly finding a solution, which further reinforces their suboptimal approach, or arriving at a dead-end and giving up. This implies that additional research is necessary on how to identify unproductive or counterproductive excessive movement with the goal of shifting students towards a learning-oriented mindset that encourages self-reflection on their processes. Additionally, strategies are needed, for example the four tracing heuristics proposed by Fitzgerald et al. (2008), which students can use and practice to help them self-assess their progress and choose a productive path forward when stuck or at a dead end.

In contrast to the lack of adoption of the explicit debugging strategies because students felt unsure about using them, another interpretation, echoed by Ko et al. (2019), is that the strategies were too advanced or sophisticated this early in their programming experience. As a result, although the strategies are valid and valuable to more experienced programmers, they may be too much when combined with also learning the programming language and turtle graphics at the same time. Although Ko et al. suggested simpler strategies and additional scaffolding for their study, I feel that the strategies in this study were already simply enough but not practiced enough due to time constraints. As a result, I feel that additional practice, including partially worked and fully worked examples using the strategies followed by in-class exercises before formal assessment, such as the debugging challenges, should be investigated further. As already noted, Mark and George both successfully used the debug print-statements, with support from the researcher, which implies that with additional practice more students could potentially integrate them effectively as well.

Finally, the third group (Lucas and Oscar) during the final challenge appeared to misunderstand the role of the debug print-statements and as a result were unable to collect the program state information necessary to debug the challenge successfully. Even with support and suggestions from the researcher, both students uncommented and ran the provided debug print-statements but then noted that they did not help with the turtle graphics output. This implies that these students misunderstood that the role of the debug print-statements was to provide information on the program state so they could identify and locate the source of the bugs. Upon reflection, although the debug

print-statements were covered throughout the course and even used to show program state during some of the code-along exercises, no assessment was undertaken of student understanding of their role and use before the final debug challenge. As a result, future instruction should incorporate student assessment of understanding and use of the strategies as well.

6.5. Limitations of the study

The primary objectives of this study were to enumerate existing student debugging strategies and challenges, and see what effects explicit instruction in debugging strategies would have on the strategies used. However, decisions were made in the design of the study that resulted in limitations on the interpretation and generalizability of the research findings.

One of the primary limitations of this study was the amount of time on task. The entire study occurred over eight weeks, with instruction taking place during the first one hour and the debugging challenges taking the second hour. As a result, students received approximately eight hours of instruction which included learning Python, turtle graphics, and the debugging strategies under study. As noted already, students showed signs of either partially learning the materials or not learning the materials, which indicates that more time was necessary for practice, assessment, and interventions to ensure they had the time necessary to successfully integrate it into their practice.

Another limitation on the interpretation of the findings stems from the fact that the majority of the data were collected via spoken concurrent think-aloud protocols in which students' spoken words and actions were recorded and later analyzed by the researcher. Although the researcher is an experienced programmer and teacher, and the coding was scrutinized by a second coder, the findings are subject to interpretation. To improve confidence in the interpretations of these findings in the future, additional coders or separate arms-length coders could be used.

In addition to uncertainties in the interpretation of the spoken-word data, students were not consistent with regard to when they spoke, what they spoke about, or the level of detail provided about their problem-solving. This is a common challenge for verbal think-aloud protocols (Bowles, 2010; Ericsson & Simon, 1993), especially when working

with younger students. In fact, Fitzgerald et al. (2010) noted that students reported finding thinking aloud distracting, and often chose not to speak. As a result, Fitzgerald et al. noted limits on the conclusions that could be drawn from some of the students' actions. They recommended developing a protocol that encouraged more speaking, and also recording the actions of the participants to provide more context, as was done in the present study. For future research, additional warm-up exercises where students think-aloud both to the researcher and to each other could help them become more fluent in thinking aloud, and perhaps come to view it as a part of their own process rather than an additional task they need to handle while also trying to solve a problem. In addition to practicing the process of thinking aloud, additional data could be collected retrospectively by interviewing students at the end of each challenge and asking them to recount or explain their approach. This recounting could then be compared to their actions and spoken statements, to provide additional context or better understand their thought process.

In addition to limits resulting from the data generation procedures, the generalizability of the results is limited by the recruitment strategy, the number of participants, and the gender of the participants. Participants in the study were all previously students in the researcher's after-school programs, and as such had all previously taken Python programming classes with the researcher before. This made recruitment easier, because the students and parents were already familiar with the researcher and the classroom setting. However, only nine participants were able to commit to the eight-week schedule for the study, and all were male. A number of additional participants (including female students) expressed interest, but could not participate due to scheduling conflicts. Although the participants were not representative of all students in this age range, their familiarity with the researcher made them more comfortable sharing their thoughts, asking questions, and generally participating in the study.

One final limitation was that the code the students debugged was written by the researcher and not by themselves. In one sense, it would be ideal for students to debug their own code, because they would not need to familiarize themselves with it and instead could focus on identifying, locating, and repairing their mistakes. However, as noted in studies that chose this approach (Katz & Anderson, 1987; Youngs, 1974), it does not ensure that all students debugged the same types of errors in the same

manner. A possible compromise was explored by Murphy et al. (2008), in which they first had students write a similar program so they developed a mental model of the program logic, and then provided them with a buggy version of the program, still written by the researchers, to debug. These and additional research designs should be considered for future research.

6.6. Suggestions for Future Research

In future studies on debugging using a concurrent think-aloud verbal protocol, it would be beneficial to augment the real-time statements with retrospective post-activity interviews, similar to Murphy et al. (2008). This would allow students who did not speak much during the activity to provide an explanation of their actions and thinking, even if it does risk the students providing rationalized responses (Bowles, 2010; Ericsson & Simon, 1993). The potential benefits are that the retrospective interview data could allow for deeper explanations by students of their strategies, and their misconceptions as well.

To increase the generalizability of findings from a similarly designed study of debugging strategies, a group of less-experienced students ("true novices"), might provide more insight into novice debugging strategies. In addition, less-experienced students may also exhibit different misconceptions and challenges that need to be identified. Working with such a group could also provide a chance to test the materials used in this study, as well as compare the effects and student responses further.

In this study, the one-on-one challenges served only as high-stakes exams rather than formative assessment tools that could help identify and correct misconceptions or failed uptake. In response to this limited up-take of the explicit strategies during this study, future researchers could create and examine the use of assessments for debugging knowledge, strategies, and misconceptions. Such assessments might help teachers identify students who required additional support in order to avoid them giving up during the challenges.

Finally, as suggested by Mathis (1974) and Lowe (2019), future research should consider more fully integrating debugging into instruction, what Lowe calls a "debugging-first" approach. This would require the creation of materials of the kind modelled by

Mathis, which led students through the process of validating their newly learned programming techniques, such that students overlearn the debugging concepts (Lowe) while completing the typical programming instruction (Mathis). This approach would complement the idea of increasing the practice of explicit debugging techniques and naturally lead towards debugging exercise and assessment as well.

6.7. Concluding Summary

This study set out to identify the strategies and challenges that novice Python programmers exhibit during a typical introductory programming course and the effects of introducing them to explicit debugging techniques during the course. Despite participants being younger than those in most prior studies of debugging, the set of strategies and challenges identified, were largely similar to those documented in other studies (Fitzgerald et al., 2008; Murphy et al., 2008). With regards to the effects of explicit debugging instruction, student uptake of the Wolf Fence algorithm and debug print-statements was limited, again matching the findings of similar studies (Böttcher et al., 2016; Ko et al., 2019). However, during the final challenge, two students showed early signs of adopting the explicit debugging strategies and through their efforts achieved the best debugging outcome. Although they needed some assistance, their efforts resulted in the correct outcome and generated further curiosity around debugging techniques.

Despite the limitations detailed above, these findings have important implications for future research and teaching. The findings suggest that providing students with explicit debugging instruction has the potential to both pique their interest in debugging and give them the tools they need to be successful. However, further research is needed to explore how to ensure that students internalize both the need for and the use of these strategies, to improve not only their debugging but also their programming as well. It is hoped that these findings provide a solid foundation for this work, so that more students can view debugging as a challenge to overcome rather than lava to be avoided.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 84–88. https://doi.org/10.1145/1067445.1067472

Allwood, C. M., & Björhag, C.-G. (1990). Novices' debugging when programming in Pascal. *International Journal of Man-Machine Studies*, *33*(6), 707–724. https://doi.org/10.1016/S0020-7373(05)80070-7

Araki, K., Furukawa, Z., & Cheng, J. (1991). A general framework for debugging. *IEEE Software*, *8*(3), 14–20. https://doi.org/10.1109/52.88939

Boies, S. J., & Gould, J. D. (1974). Syntactic Errors in Computer Programming. *Human Factors*, *16*(3), 253–257. https://doi.org/10.1177/001872087401600307

Bonar, J. (1985). *Understanding the Bugs of Novice Programmers* (COINS Technical Report 85–12; p. 81). University of Massachusetts.

Böttcher, A., Thurner, V., Schlierkamp, K., & Zehetmeier, D. (2016). Debugging students' debugging process. *2016 IEEE Frontiers in Education Conference (FIE)*, 1–7. https://doi.org/10.1109/FIE.2016.7757447

Bowles, M. A. (2010). *The Think-Aloud Controversy in Second Language Research*. Routledge. https://doi.org/10.4324/9780203856338

Brooks, R. E. (1980). Studying programmer behavior experimentally: The problems of proper methodology. *Communications of the ACM*, *23*(4), 207–213. https://doi.org/10.1145/358841.358847

Brown, J. S., & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, *4*(4), 379–426. https://doi.org/10.1016/S0364-0213(80)80010-3

Carver, S. M. (1986). *Transfer of LOGO Debugging Skill: Analysis, Instruction, and Assessment* [Doctoral Dissertation, Carnegie-Mellon University]. https://eric.ed.gov/?id=ED284678

Carver, S. M., & Klahr, D. (1986). Assessing children's LOGO debugging skills with a formal model. *Journal of Educational Computing Research*, *2*(4), 487–525. https://doi.org/10.2190/KRD4-YNHH-X283-3P

Carver, S. M., & Risinger, S. C. (1987). Improving children's debugging skills. In *Empirical studies of programmers: Second workshop* (pp. 141–171). https://dl.acm.org/doi/abs/10.5555/54968.54978

Charmaz, K. (2006). *Constructing grounded theory: A practical guide through qualitative analysis*. Sage Publications.

Chiu, I., & Shu, L. H. (2011). *Potential Limitations of Verbal Protocols in Design Experiments*. 287–296. https://doi.org/10.1115/DETC2010-28675

Chmiel, R., & Loui, M. C. (2003). An integrated approach to instruction in debugging computer programs. *33rd Annual Frontiers in Education, 2003. FIE 2003.*, *3*, S4C-1. https://doi.org/10.1109/FIE.2003.1266016

Chmiel, R., & Loui, M. C. (2004). Debugging: From Novice to Expert. *ACM SIGCSE Bulletin*, *36*(1), 17–21. https://doi.org/10.1145/1028174.971310

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, *20*(1), 37–46. https://doi.org/10.1177/001316446002000104

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. *Proceedings of the Fourteenth Australasian Computing Education Conference*, *123*, 77–86. https://dl.acm.org/doi/10.5555/2483716.2483726

D'souza, R., Bhayana, M., Ahmadzadeh, M., & Harrington, B. (2019). A Mixed-Methods Study of Novice Programmer Interaction with Python Error Messages. *Proceedings of the Western Canadian Conference on Computing Education*, 1–2. https://doi.org/10.1145/3314994.3325090

Dweck, C. S., & Elliott, E. S. (1983). Achievement Motivation. In *Handbook of child psychology: Social and personality development.* (pp. 643–691). Wiley.

Ericsson, K. A., & Simon, H. A. (1980). Verbal Report as Data. *Psychological Review*, *87*(3), 215–251. https://doi.org/10.1037/0033-295X.87.3.215

Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis: Verbal reports as data* (Revised Edition). The MIT Press.

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. (1969). *Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final Report on the First Fifteen Months of the LOGO Project*. https://eric.ed.gov/?id=ED038034

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging From the Student Perspective. *IEEE Transactions on Education*, *53*(3), 390–396. https://doi.org/10.1109/TE.2009.2025266

Fitzgerald, S., Simon, B., & Thomas, L. (2005). Strategies that students use to trace code: An analysis based in grounded theory. *Proceedings of the First International Workshop on Computing Education Research*, 69–80. https://doi.org/10.1145/1089786.1089793

Frankish, K. (2010). Dual-Process and Dual-System Theories of Reasoning. *Philosophy Compass*, *5*(10), 914–926. https://doi.org/10.1111/j.1747-9991.2010.00330.x

Gauss, E. J. (1982). The “Wolf Fence” algorithm for debugging. *Communications of the ACM*, *25*(11), 780. https://doi.org/10.1145/358690.358695

Gilmore, D. J. (1991). Models of debugging. *Acta Psychologica*, *78*, 151–172. https://doi.org/10.1016/0001-6918(91)90009-O

Gould, J. D. (1975). Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies*, *7*, 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8

Gould, J. D., & Drongowski, P. (1974). An Exploratory Study of Computer Program Debugging. *Human Factors*, *16*(3), 258–277. https://doi.org/10.1177/001872087401600308

Gugerty, L., & Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. *Empirical Studies of Programmers: First Workshop*, 13–27. https://dl.acm.org/doi/abs/10.5555/21842.28883

Hays, D. G., & Singh, A. A. (2011). *Qualitative Inquiry in Clinical and Educational Settings*. Guilford Press.

Heilman, R. L., & Ashby, G. P. (1971). Re-evaluation of debugging in the computer science curriculum. *ACM SIGCSE Bulletin*, *3*(4), 15–18. https://doi.org/10.1145/382214.382215

Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. *AERA Annual Meeting*, 1–17.

Johnson, M. S. (1982). A software debugging glossary. *ACM SIGPLAN Notices*, *17*(2), 53–70. https://doi.org/10.1145/947902.947908

Kahneman, D. (2013). *Thinking, Fast and Slow*. Farrar, Straus and Giroux. https://us.macmillan.com/books/9780374533557/thinkingfastandslow

Katz, I. R., & Anderson, J. R. (1987). Debugging: An Analysis of Bug-Location Strategies. *Human–Computer Interaction*, *3*(4), 351–399. https://doi.org/10.1207/s15327051hci0304_2

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in LISP. *Empirical Studies of Programmers: First Workshop*, 198–212.

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, *20*(3), 362–404. https://doi.org/10.1016/0010-0285(88)90004-7

Ko, A. J., LaToza, T. D., Hull, S., Ko, E. A., Kwok, W., Quichocho, J., Akkaraju, H., & Pandit, R. (2019). Teaching Explicit Programming Strategies to Adolescents. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 469–475. https://doi.org/10.1145/3287324.3287371

Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, *16*(1), 41–84. https://doi.org/10.1016/j.jvlc.2004.08.003

Kocher, W. (1969). *A survey of current debugging concepts* (Contractor Report (CR) NASA-CR-1397; p. 98). NASA. https://ntrs.nasa.gov/api/citations/19690026235/downloads/19690026235.pdf

Lewis, C. M. (2012). The importance of students' attention to program state: A case study of debugging behavior. *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, 127–134. https://doi.org/10.1145/2361276.2361301

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150. https://doi.org/10.1145/1041624.1041673

Litecky, C. R., & Davis, G. B. (1976). A study of errors, error-proneness, and error diagnosis in Cobol. *Communications of the ACM*, *19*(1), 33–38. https://doi.org/10.1145/359970.359991

Lowe, T. (2019). Debugging: The key to unlocking the mind of a novice programmer? *2019 IEEE Frontiers in Education Conference (FIE)*, 1–9. https://doi.org/10.1109/FIE43999.2019.9028699

Lukey, F. J. (1980). Understanding and debugging programs. *International Journal of Man-Machine Studies*, *12*(2), 189–202. https://doi.org/10.1016/S0020-7373(80)80017-4

Mathis, R. F. (1974). Teaching debugging. *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*, 59–63. https://doi.org/10.1145/800183.810443

McCartney, R., Eckerdal, A., Moström, J. E., Sanders, K., & Zander, C. (2007). Successful students' strategies for getting unstuck. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 156–160. https://doi.org/10.1145/1268784.1268831

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, 125–180. https://doi.org/10.1145/572133.572137

Merrill, M. D. (2002). First principles of instruction. *Educational Technology Research and Development*, *50*(3), 43–59. https://doi.org/10.1007/BF02505024

Michaeli, T., & Romeike, R. (2019a). Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. *2019 IEEE Global Engineering Education Conference (EDUCON)*, 1030–1038. https://doi.org/10.1109/EDUCON.2019.8725282

Michaeli, T., & Romeike, R. (2019b). Improving Debugging Skills in the Classroom—The Effects of Teaching a Systematic Debugging Process. *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, 1–7. https://doi.org/10.1145/3361721.3361724

Miles, Matthew. B., & Huberman, A. M. (1994). *Qualitative data analysis: An expanded sourcebook* (2nd Edition). SAGE Publications.

Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies*, *6*(2), 237–260. https://doi.org/10.1016/S0020-7373(74)80004-0

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: The good, the bad, and the quirky -- a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, *40*(1), 163–167. https://doi.org/10.1145/1352322.1352191

Myers, G. J. (1978). A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, *21*(9), 760–768. https://doi.org/10.1145/359588.359602

Nanja, M. (1988). *An Investigation of the On-line Debugging Process of Expert and Novice Student Programmers* [PhD Thesis]. Oregon State University.

Nanja, M., & Cook, C. R. (1987). An analysis of the on-line debugging process. In *Empirical Studies of Programmers: Second Workshop* (pp. 172–184). Ablex. https://dl.acm.org/doi/abs/10.5555/54968.54979

Newell, A., & Simon, H. A. (1972). *Human problem solving* (pp. xiv, 920). Prentice-Hall.

Nisbett, R. E., & Wilson, T. D. (1977). Telling More Than We Can Know: Verbal Reports on Mental Processes. *Psychological Review*, *84*(3), 231–259. https://doi.org/10.1037/0033-295X.84.3.231

O'Dell, D. H. (2017). The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, *15*(1), 71–90. https://doi.org/10.1145/3055301.3068754

Papert, S. (1972). Teaching Children Thinking. *Programmed Learning and Educational Technology*, *9*(5), 245–255. https://doi.org/10.1080/1355800720090503

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.

Papert, S., & Solomon, C. (1971). *Twenty things to do with a computer* (Memo A.I. Memo No. 248; Logo Memo No. 3.). Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/5836

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, *2*(1), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 213–229.

Rich, K. M., Strickland, C., Binkowski, T. A., & Franklin, D. (2019). A K-8 Debugging Learning Trajectory Derived from Research Literature. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 745–751. https://doi.org/10.1145/3287324.3287396

Ripley, G. D., & Druseikis, F. C. (1978). A statistical analysis of syntax errors. *Computer Languages*, *3*(4), 227–240. https://doi.org/10.1016/0096-0551(78)90041-3

Sheil, B. A. (1981). The Psychological Study of Programming. *ACM Computing Surveys (CSUR)*, *13*(1), 101–120. https://doi.org/10.1145/356835.356840

Shneiderman, B., & McKay, D. (1976). Experimental Investigations of Computer Program Debugging and Modification. *Proceedings of the Human Factors Society Annual Meeting*, *20*(24), 557–563. https://doi.org/10.1177/154193127602002401

So, M. H., & Kim, J. M. (2018). An analysis of the difficulties of elementary school students in Python programming learning. *International Journal on Advanced Science, Engineering and Information Technology*, *8*(4–2), 1507–1512. https://doi.org/10.18517/ijaseit.8.4-2.2720

Spohrer, J. C., & Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. *Empirical Studies of Programmers: First Workshop*, 230–251. https://dl.acm.org/doi/abs/10.5555/21842.28897

van Merriënboer, J. J. G., Clark, R. E., & de Croock, M. B. M. (2002). Blueprints for complex learning: The 4C/ID-model. *Educational Technology Research and Development*, *50*(2), 39–61. https://doi.org/10.1007/BF02504993

Vessey, I. (1984). *An investigation of the psychological processes underlying the debugging of computer programs.* [Doctoral Dissertation]. University of Queensland.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494. https://doi.org/10.1016/S0020-7373(85)80054-7

Vessey, I. (1986). Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, *16*(5), 621–637. https://doi.org/10.1109/TSMC.1986.289308

Wentworth, P., Elkner, J., Downey, A. B., & Meyers, C. (2012). *How to Think Like a Computer Scientist: Learning with Python 3* (3rd Edition). https://openbookproject.net/thinkcs/python/english3e/

Wescourt, K. T., & Hemphill, L. (1978). *Representing and Teaching Knowledge for Troubleshooting/Debugging. Technical Report No. 292*. Institute for Mathematical Studies in the Social Sciences, Stanford University. https://eric.ed.gov/?id=ED152321

Whalley, J., Settle, A., & Luxton-Reilly, A. (2023). A Think-Aloud Study of Novice Debugging. *ACM Transactions on Computing Education*, *23*(2), 28:1-28:38. https://doi.org/10.1145/3589004

Youngs, E. A. (1974). Human Errors in Programming. *International Journal of Man-Machine Studies*, *6*(3), 361–376. https://doi.org/10.1016/S0020-7373(74)80027-1

Appendix A.

Code Book

Fall 2022 Debugging Study - Coders Manual (Version 4)

The aim of this research project is to investigate the strategies that novice programmers use while debugging computer programs. To investigate these strategies, participants were asked to think-aloud while they attempted to debug computer programs and their statements and actions were recorded. These recording were then transcribed into a series of short, numbered phrases, which according to Newell & Simon (1972) represent "a naïve assessment of what constitutes a single take assertion or reference by the subject" (Vessey, 1984, p. 427). These short phrases then need to be encoded using the codes and heuristics described in this code book.

How to Encode

To encode a transcribed debugging session, first you will need the session transcription and a debugging session coder spreadsheet (Figure A1). For each of the lines of the transcript that contains a debugging action, choose one of the codes from the drop-down list in column F. If none of the codes available match the action, choose the No-Match option.

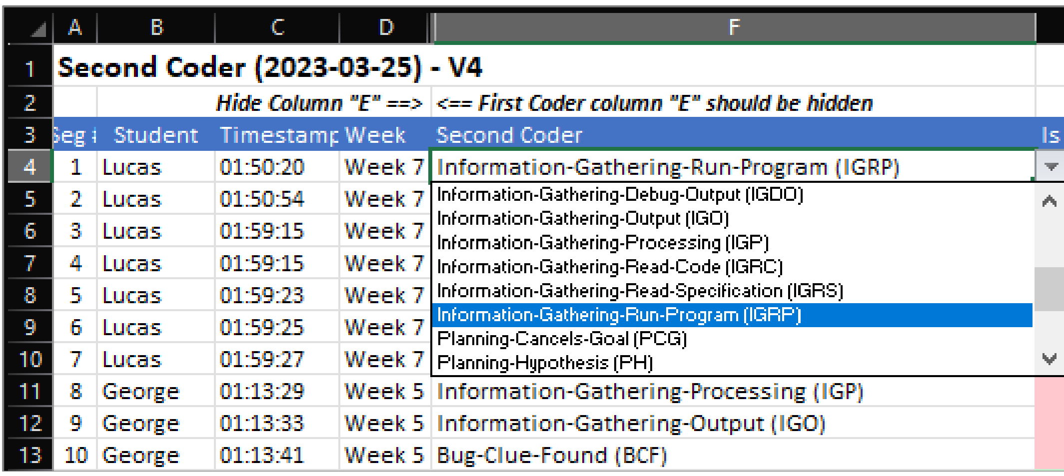


Figure A1. Sample debugging session coder spreadsheet.

Coding Heuristics

The following are suggestions to follow when encoding specific situations where the choice of codes may be ambiguous.

Order of precedence when encountering multiple possible codes:

In situations where multiple codes are applicable, I have, where possible, separated the transcription lines into multiple distinct lines to support individual coding of individual situations or actions. However, in situations where it is unclear which of a set of possible competing codes *from a different category* to choose, then the following order of precedence is to be used. Planning codes are chosen over both Bug Repair and Information Gathering. As well, Bug Repair codes are chosen over Information Gathering codes.

More simply: Planning > Bug Repair > Information Gathering

Note that Self-Efficacy is not included in the order of precedence above because those comments are asked and answered separately from the debugging codes.

Common Planning versus Bug Report Situation

The most common situation where a planning code is possible, and a bug report code is possible is at the end of a debugging session where a student has completed or abandoned a planned or hypothesized scenario. In this case the student will first need to verify that their change was successful or not and then verify if they have completed the task or not. This results in a Bug-Report-Verify (BRV) and either a Planning-Satisfies-Goal (PSG) or Planning-Cancels-Goal (PCG) depending on the outcome of their change. The planning codes should be used when the student finishes or abandons the bug and the verify should be used when they are still working on it.

Order of precedence for the run program rules (BRRP & IGRP).

There are two separate "run-program" codes: Bug-Repair-Run-Program (BRRP) and Information-Gathering-Run-Program (IGRP). BRRP is only used when the student runs their program immediately after making a bug repair so they can try to validate the results. IGRP typically happens at the start of the debugging session when the student

wants to see what the program output looks like before attempting any repairs or during their debugging session when they just want to see the output but have not made any bug repairs related to this program run.

Coding Categories

The codes are grouped into four coding categories: planning, bug related, information gathering, and self-efficacy.

Planning (PL)

Planning activities happen when the student states that they are going to focus on a particular problem or sub-problem (Planning-Set-Goal) or when the student states that they think something is the problem. Planning is split into three situations: (1) stating an explicit goal (Planning-Set-Goal), (2) stating a hypothesis (Planning-Hypothesis), or (3) reaching the end of a goal by achieving the stated goal (Planning-Satisfies-Goal); or cancelling the goal implicitly or explicitly by abandoning or switching tasks (Planning-Cancel-Goal).

Name	**Description**
Planning-Set-Goal (PSX)	When the student states the start of a goal or a particular end state. For example, “I’m going to start with the green square” or “Okay, let’s look at the line thickness.”
Planning-Cancels-Goal (PCG)	When the student cancels a previous goal either stating it explicitly or switching implicitly by starting a new task. For example, when they abandon one idea and instead switch to another target or goal.
Planning-Satisfies-Goal (PSG)	When the previously set goal (PSX) has been successfully achieved either implicitly by their actions or stated explicitly by the student. For example: “That fixed it,” “That got it,” “There it is,” etc.
Planning-Hypothesis (PH)	When the student states a hypothesis about a bug and includes possible code-specific or code-related reasons. For example, "I think this is happening because they didn't put the code into functions."

Q: When is it PSX vs PH?

A: When in doubt between PH and PSX, PSX needs to be an explicit goal or target while PH is more an expression that either already is or can be prefaced with “I think.” For example, if the student says they are going to look at something specifically then that is probably PSX but when they imply or say they think that something might be the cause or worth looking into then that is a hypothesis (PH).

Bug Related (BR)

Bug-related activities occur when the student is attempting to find, repair, or verify a bug.

Name	Description
Bug-Clue-Found (BCF)	When the student finds a clue. This code is assigned when the participant discovers a problem with the program, i.e., that the program is not performing as it should. The CF code is assigned only once for each clue (use CS when additional statements, evidence, or references to this clue are made). BCF is only used when it is made about the code itself and not about an issue with the output.
Bug-Observation (BO)	When the student makes an observation about a bug without providing a reason or guess. For example, "There is something wrong with the blue square."
Bug-Repair-Correct (BRC)	When the participant correctly repairs a bug by making a correct change to the code. Does not include undoing code (see BRU instead).
Bug-Repair-Incorrect (BRI)	When the student makes an incorrect change to the program while attempting to fix a bug.
Bug-Repair-Run-Program (BRRP)	When the student runs the program after a repair in an effort to verify the repair.
Bug-Repair-Undo (BRU)	When the student undoes a coding line change that they introduced. For example, if they remove a line they previously added or adds back a line they previously deleted. As well, if a student restores a line back to a previous state, for example changing the value of a variable from A to B and then restoring it back to A.
Bug-Repair-Verify (BRV)	When the student compares or verifies their output against the target output either by stating a comparison or comparison result. For example, "Now they match" or "Oh what, why is that not the same now?"

Q: When is a change an undo (BRU) rather than a BRC/BRI?

A: BRU is only used when the student restores the code back to its previous state either using a keyboard shortcut like CTRL-Z or by restoring the previous value. If the student instead changes the value to a new value, either the correct or incorrect value then it is a BRC or BRI instead.

Information Gathering (IG)

Information gathering refers to activities in which the participant seeks information from the task materials, i.e., the code, assignment specifications, and program inputs/outputs.

Name	**Description**
Information-Gathering-Code-Editor-Hint (IGCEH)	Student has acted on or commented on a hint from the code editor. For example, red underline for errors and grey or yellow highlights or underlines for warnings. If the student appears to or explicitly states that they are making a decision because they saw an editor hint and are acting upon it. For example, “That line has a red error underline, so I’m going to look at it.”
Information-Gathering-Compiler-Error-Message (IGCEM)	When the student reads the compiler error message or runtime error message in the output window. This only happens after the student has attempted to run their program and either before the program runs or during its running it fails and reports an error in the output window in the form of a traceback compiler error message.
Information-Gathering-Debug-Output (IGDO)	When the student reads the debug output from a debug print-statement in the output window. Note that this is only for output from a debug print-statement that either the student added to the code or that was part of the original code. Output that is read that is not from a debug print-statement is covered by IGO instead.
Information-Gathering-Output (IGO)	When the student reads non-debugging output from the output window. For example, when the read the output from a print statement that is the result of running the program, but which is not specifically for debugging the program. Also, output does not include compiler or runtime error message, use “IGCEM” for those instead.
Information-Gathering-Processing (IGP)	When the participant shows evidence of mentally processing or tracing the program and can include the establishment of a point in processing from which to commence.
Information-Gathering-Read-Code (IGRC)	When the student either explicitly reads the code as-is out loud or implicitly appears to read the program source code by moving their mouse over and around the source code. Note that reading the code aloud must mostly match the actual syntax. If the student is instead reading part of the code but appears to be synthesizing the code values or flow, then they are processing the code and that should instead be IGP.
Information-Gathering-Read-Specification (IGRS)	When the student reads or refers to the program or challenge specification document for information.
Information-Gathering-Run-Program (IGRP)	The participant has run the program to see what it does. This is not the same as running the program to test a code change or to observe debugging information (BRRP).

Self-Efficacy (SE)

Self-efficacy codes occur when the students make a statement about their confidence related to debugging the code. Student self-efficacy was measured via pre- and post-session questions about their confidence debugging on a scale from one to seven with one being the lowest and seven the highest. Following the question of confidence scale, students were then asked why they felt it went up, down, or stayed the same relative to their previously reported value.

Name	**Description**
Self-Efficacy-Less-Confidence (SELC)	When the student makes a comment about something lowering their debugging confidence. For example, "I'm feeling less confident now."
Self-Efficacy-More-Confidence (SEMC)	When a student makes a comment about something improving or raising their debugging confidence. For example, "I feel more confident after debugging that code."
Self-Efficacy-No-Change (SENC)	When the student states that their confidence did not change as a result of the just completed debugging session. For example, "Q: Why no change to your confidence?", "A: Nothing changed, I still feel the same level of confidence."

Appendix B.

Weekly Challenge Python Code

Week 1 Debugging Session Python Code (Challenges 1-4)

Original URL: https://www.xmodus.com/assets/sfu-study-fall-2022/week1-challenges.txt

Week 1 – Challenge #1 – 2x Syntax Errors

Bugged Version:

```
1  # Challenge #1
2  age = 1
3  age = Age + 1
4  print(a)
```

Number of bugs: 2

- Line 3: NameError: “Age” is not defined; should be “age”.
- Line 4: NameError: name ‘a’ is not defined; should be “age”.

Corrected Version:

```
1  # Challenge #1
2  age = 1
3  age = age + 1
4  print(age)
```

Week 1 – Challenge #2 – 1x Syntax Error

Bugged Version:

```
1  # Challenge #2
2  name = "Billy"
3  PRINT(name)
```

Number of bugs: 1

- Line 3: NameError: name ‘PRINT’ is not defined, should be “print”.

Corrected Version:

1	`# Challenge #2`
2	`name = "Billy"`
3	`print(name)`

Week 1 – Challenge #3 – 1x Indentation Error

Bugged Version:

1	`# Challenge #3`
2	`age = 6`
3	`after_birthday = age + 1`
4	`  print(after_birthday)`

Number of bugs: 1

- *IndentationError: unexpected indent [Line 4]; leading spaces need to be removed.*

Corrected Version:

1	`# Challenge #3`
2	`age = 6`
3	`after_birthday = age + 1`
4	`print(after_birthday)`

Week 1 – Challenge #4 – 1x Semantic Error

Bugged Version:

```
1  # Challenge #4
2  height = 2
3  height = height + 2
4  height + 2
5  # at the end, the height should be 6
6  print(height)
```

Number of bugs: 1

- Line 4: This statement, “height + 2” does not modify the height and instead is considered a “null operation”. As a result, height returns four (4) instead of six (6). This should be “height = height + 2”, like line 3, or the equivalent expression “height += 2”.

Corrected Version 1:

```
1  # Challenge #4
2  height = 2
3  height = height + 2
4  height = height + 2
5  # at the end, the height should be 6
6  print(height)
```

Corrected Version 2:

```
1  # Challenge #4
2  height = 2
3  height = height + 2
4  height += 2
5  # at the end, the height should be 6
6  print(height)
```

Week 4/5 Debugging Sessions Python Code (4x semantic errors)

Bugged Version:

```
1   # 3 nested squares - variables and loops (bugged)
2   from turtle import *
3   setup(500, 500)
4
5   penup()
6   goto(-50, 50)
7   pendown()
8
9   square_size = 30
10  pen_size = 3
11
12  print("square 1")
13  pencolor("red")
14  pensize(pen_size)
15  for i in range(4):
16      x = square_size
17      forward(x)
18      print("i=", i, "x=", x, "square_size=", square_size)
19      right(90)
20
21  square_size += 30
22  print("square 2")
23  pencolor("green")
24  pensize(1)
25  for j in range(4):
26      x = -square_size
27      forward(x)
28      print("j=", j, "x=", x, "square_size=", square_size)
29      right(90)
30
31  square_size += 30
32  print("square 3")
33  pencolor("blue")
34  pensize(1)
35  for k in range(1, 4):
36      x = square_size
37      forward(x)
38      print("k=", k, "x=", x, "square_size=", square_size)
39      right(90)
40
41  done()
```

Number of bugs: 4

- Line 24: Incorrect value; should be pensize(pen_size).

- Line 26: Logic error; should be “x = square_size”
- Line 34: Incorrect value; should be pensize(pen_size).
- Line 35: Off-by-one error; should be “for k in range(4)”, “for k in range(1, 5)”, or “for k in range(0, 4)”.

Corrected Version:

```
1   # 3 nested squares - variables and loops (bugged)
2   from turtle import *
3   setup(500, 500)
4
5   penup()
6   goto(-50, 50)
7   pendown()
8
9   square_size = 30
10  pen_size = 3
11
12  print("square 1")
13  pencolor("red")
14  pensize(pen_size)
15  for i in range(4):
16      x = square_size
17      forward(x)
18      print("i=", i, "x=", x, "square_size=", square_size)
19      right(90)
20
21  square_size += 30
22  print("square 2")
23  pencolor("green")
24  pensize(pen_size)
25  for j in range(4):
26      x = square_size
27      forward(x)
28      print("j=", j, "x=", x, "square_size=", square_size)
29      right(90)
30
31  square_size += 30
32  print("square 3")
33  pencolor("blue")
34  pensize(pen_size)
35  for k in range(4):
36      x = square_size
37      forward(x)
38      print("k=", k, "x=", x, "square_size=", square_size)
39      right(90)
40
41  done()
```

Week 7/8 Debugging Sessions Python Code (2x logic errors)

Bugged Version:

```
1   # week7_challenge.py
2   # this script should draw a series of nested diamonds, in
    the sequence: green smallest, red middle, blue biggest
3   from turtle import *
4   setup(800, 600)
5
6   colours = ["red", "green", "blue"]
7
8   def square(size):
9       for i in range(4):
10          forward(size)
11          right(90)
12
13  def is_even(n):
14      if n % 2 == 0:
15          return False
16      else:
17          return True
18
19  # reposition
20  penup()
21  goto(-100, 0)
22  pendown()
23
24  # start drawing
25  pencolor("black")
26  pensize(2)
27  left(45)
28
29  for s in range(5, 0, -1):
30
31      # get the color
32      if is_even(s):
33          fc = "white"
34      else:
35          fc = colours[s % 3]
36
37      # some debugging print statements
38      # print("s", s)
39      # print("s % 3", s, s % 3)
40      # print("is_even", s, is_even(s))
41      # print("fc", fc)
42
43      # draw the square
44      # pencolor(fc)
45      fillcolor(fc)
```

```
46        begin_fill()
47        square(s * 30)
48        end_fill()
49
50    # hideturtle()
51
52    done()
```

Number of bugs: 2

- The “is_even” function returns True when the number if odd and False when the number is even. This is because Line 14 should be “if n % 2 == **1**”, not “if n % 2 == 0”. Alternatively, the return values on lines 15 & 17 could be swapped so that is_even returns the correct True for even number and False for odd numbers. Although the is_even() function should be corrected, an alternative would be to negate the return value for the function on line 32, changing it to “if not is_even(s)” or swapping the order of the if-statement lines 33 and 35 to accomplish the same result.

- Line 44 should be uncommented so that the outline lines for the squares are the same color as the fill color rather than the default black.

Corrected Version:

```
1     # week7_challenge.py
2     # this script should draw a series of nested diamonds, in
      the sequence: green smallest, red middle, blue biggest
3     from turtle import *
4     setup(800, 600)
5
6     colours = ["red", "green", "blue"]
7
8     def square(size):
9         for i in range(4):
10            forward(size)
11            right(90)
12
13    def is_even(n):
14        if n % 2 == 1:
15            return False
16        else:
17            return True
18
19    # reposition
20    penup()
21    goto(-100, 0)
```

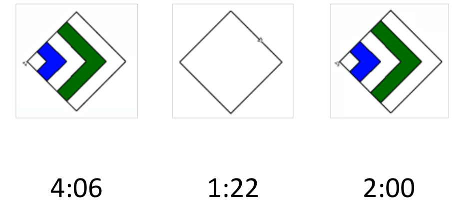
```
22  pendown()
23
24  # start drawing
25  pencolor("black")
26  pensize(2)
27  left(45)
28
29  for s in range(5, 0, -1):
30
31      # get the color
32      if is_even(s):
33          fc = "white"
34      else:
35          fc = colours[s % 3]
36
37      # some debugging print statements
38      # print("s", s)
39      # print("s % 3", s, s % 3)
40      # print("is_even", s, is_even(s))
41      # print("fc", fc)
42
43      # draw the square
44      pencolor(fc)
45      fillcolor(fc)
46      begin_fill()
47      square(s * 30)
48      end_fill()
49
50  # hideturtle()
51
52  done()
```

Appendix C.

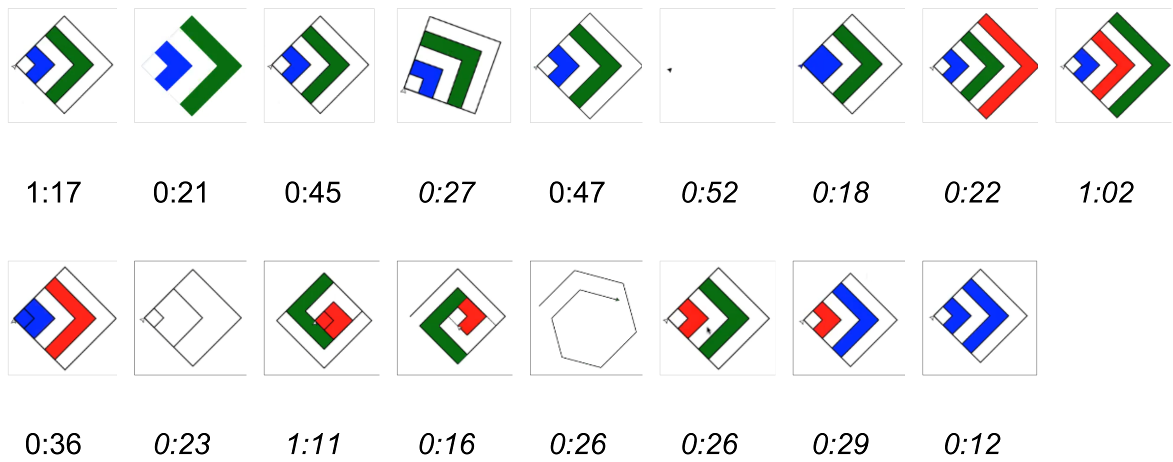
Student Debug Output Progression (Week 7/8)

The following are the graphical outputs that resulted when the students ran the Week 7/8 debugging challenge. These are ordered by the sequence students participated in the challenge, i.e., Oscar was the first student, then Lucas, etc. The times below each image are relative to the previous run time or the start of the session for the first image. This is meant to show how long students took between each run of the program.

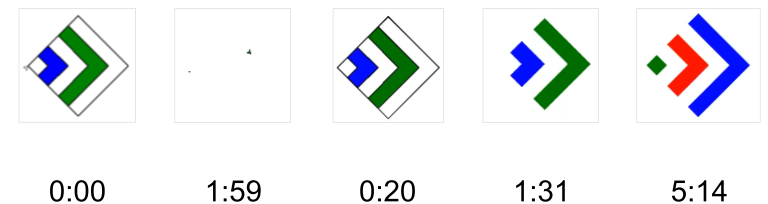
Oscar (S7) – 3 runs – Total time: 8:01 (unsuccessful).



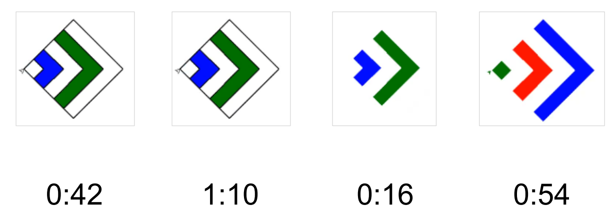
Lucas (S5) – 17 runs – Total time: 10:20 (unsuccessful).



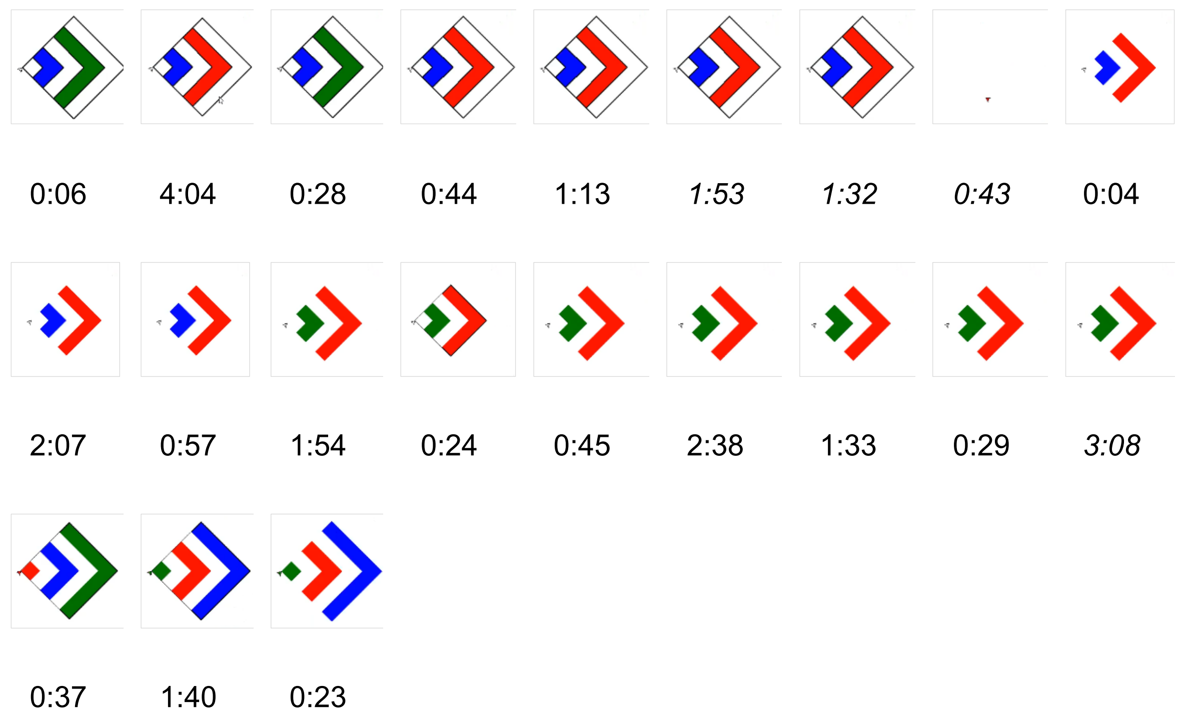
Brian (S1) – 5 runs – Total time: 9:57 (successful).



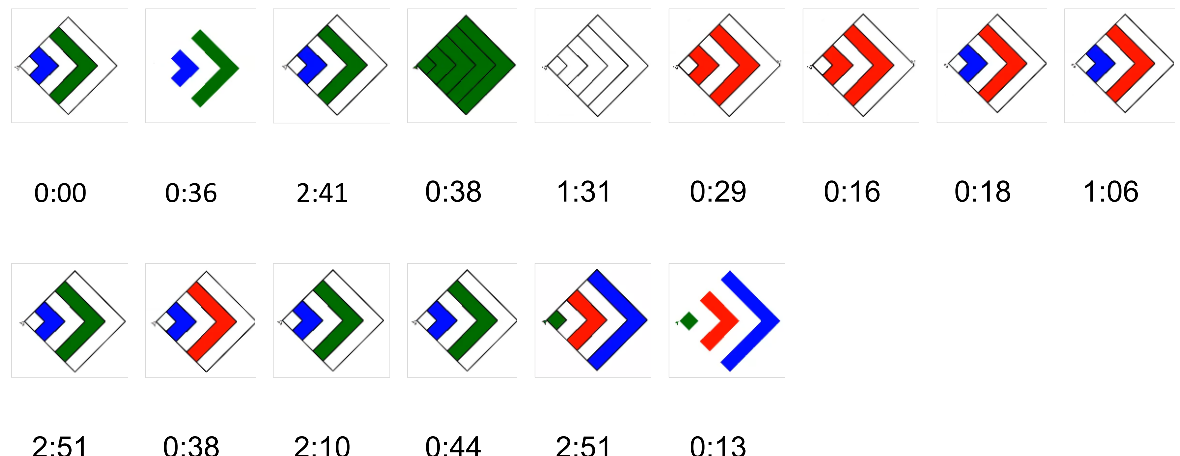
Peter (S8) – 4 runs – Total time: 3:20 (successful).



Mark (S6) – 21 runs – Total time: 27:40 (successful).



George (S3) – 12 runs – Total time: 19:33 (successful).



Appendix D.

Debugging Session Episodes

Weeks 4/5

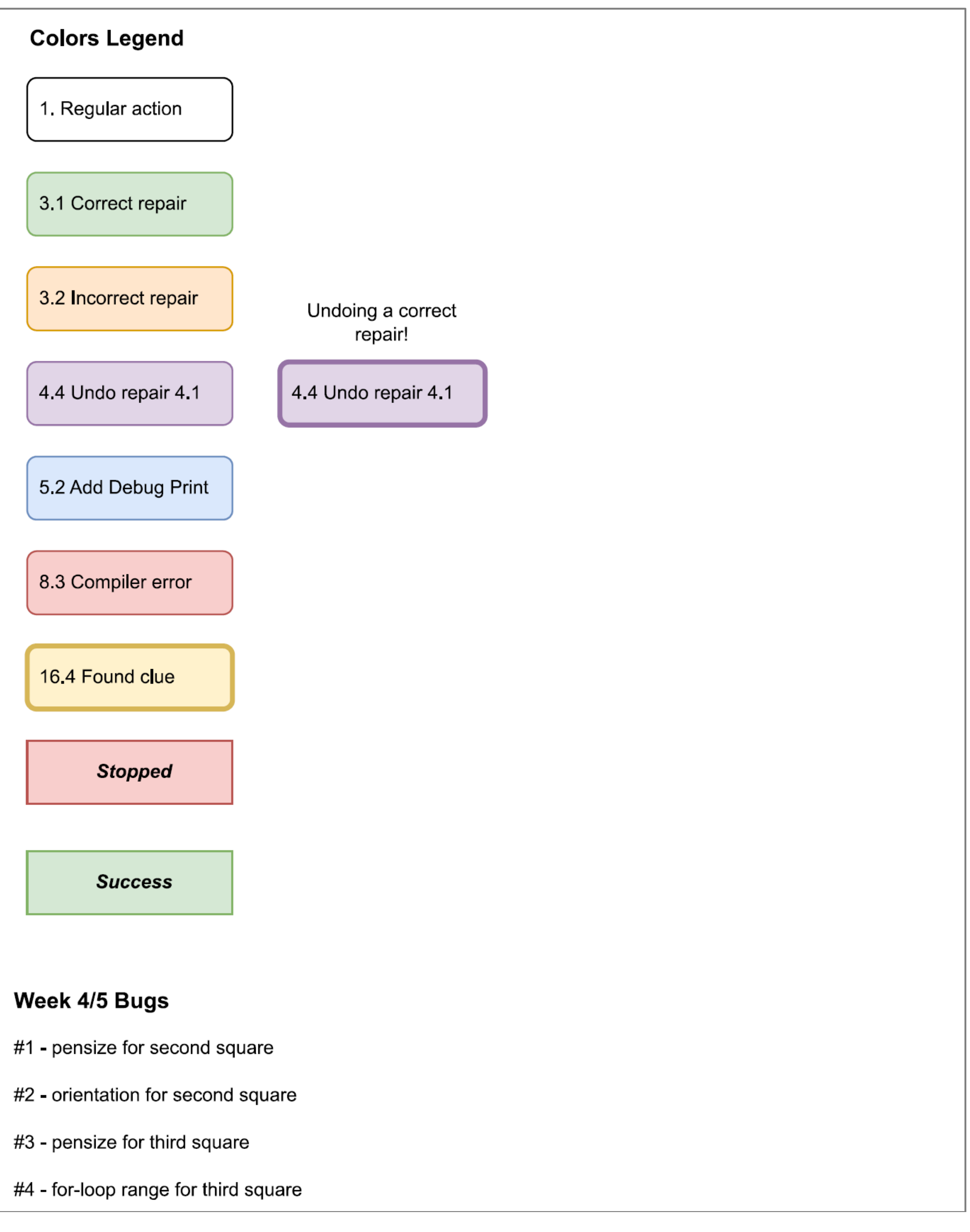


Figure D1. Week 4/5 Debugging Session Episode Colors Legend and Bugs List

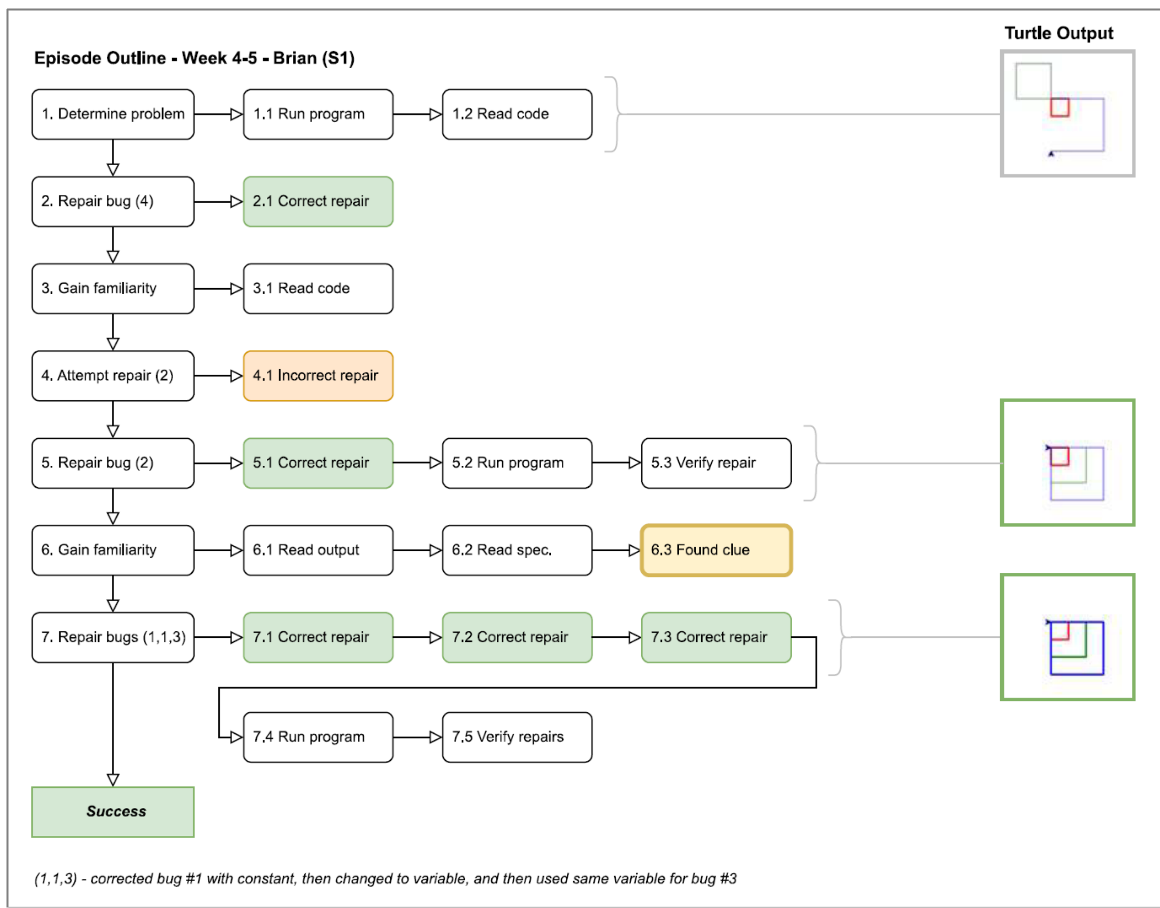


Figure D2. Week 4/5 – Brian’s debugging session episodes.

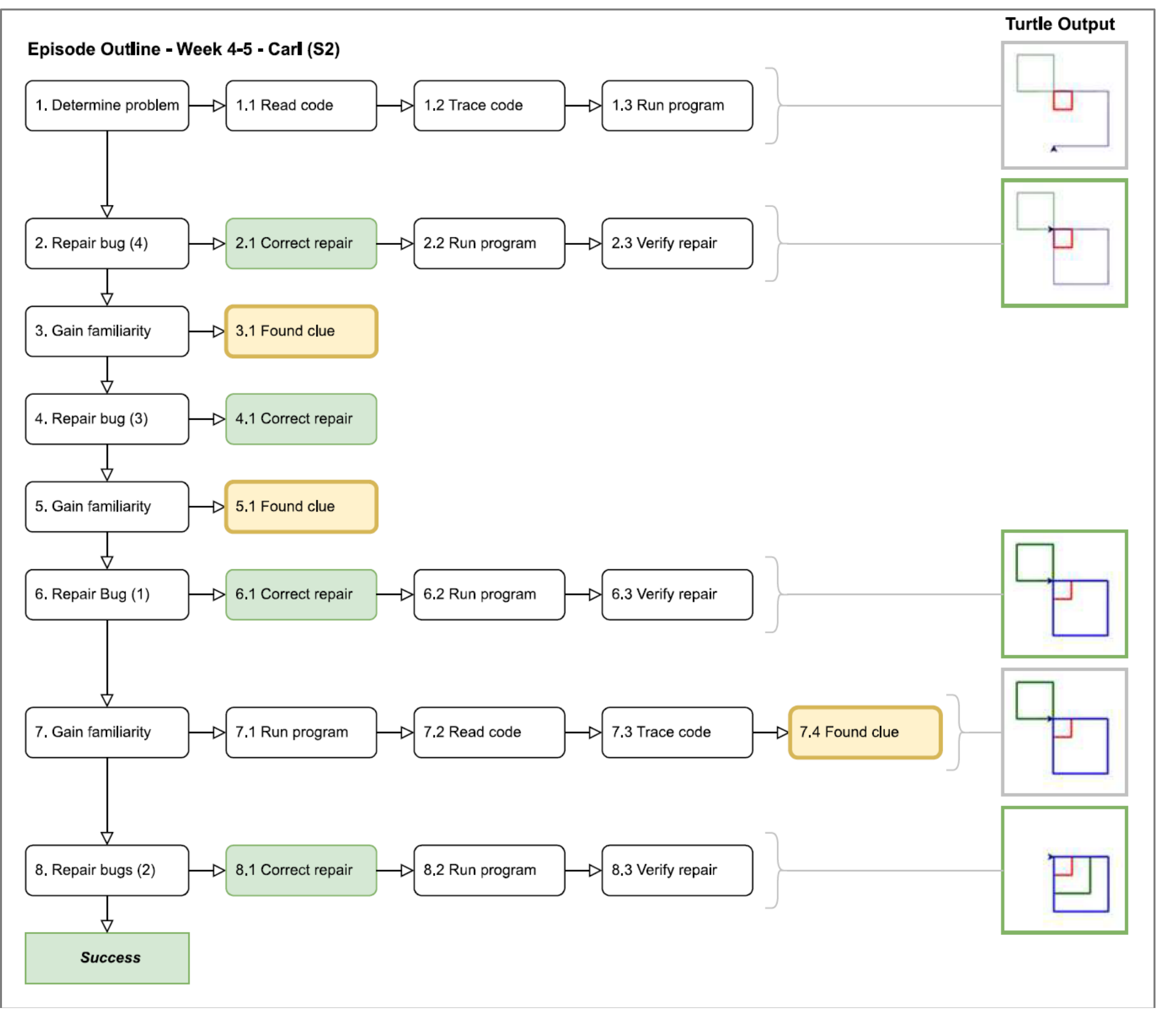


Figure D3. **Week 4/5 – Carl's debugging session episodes.**

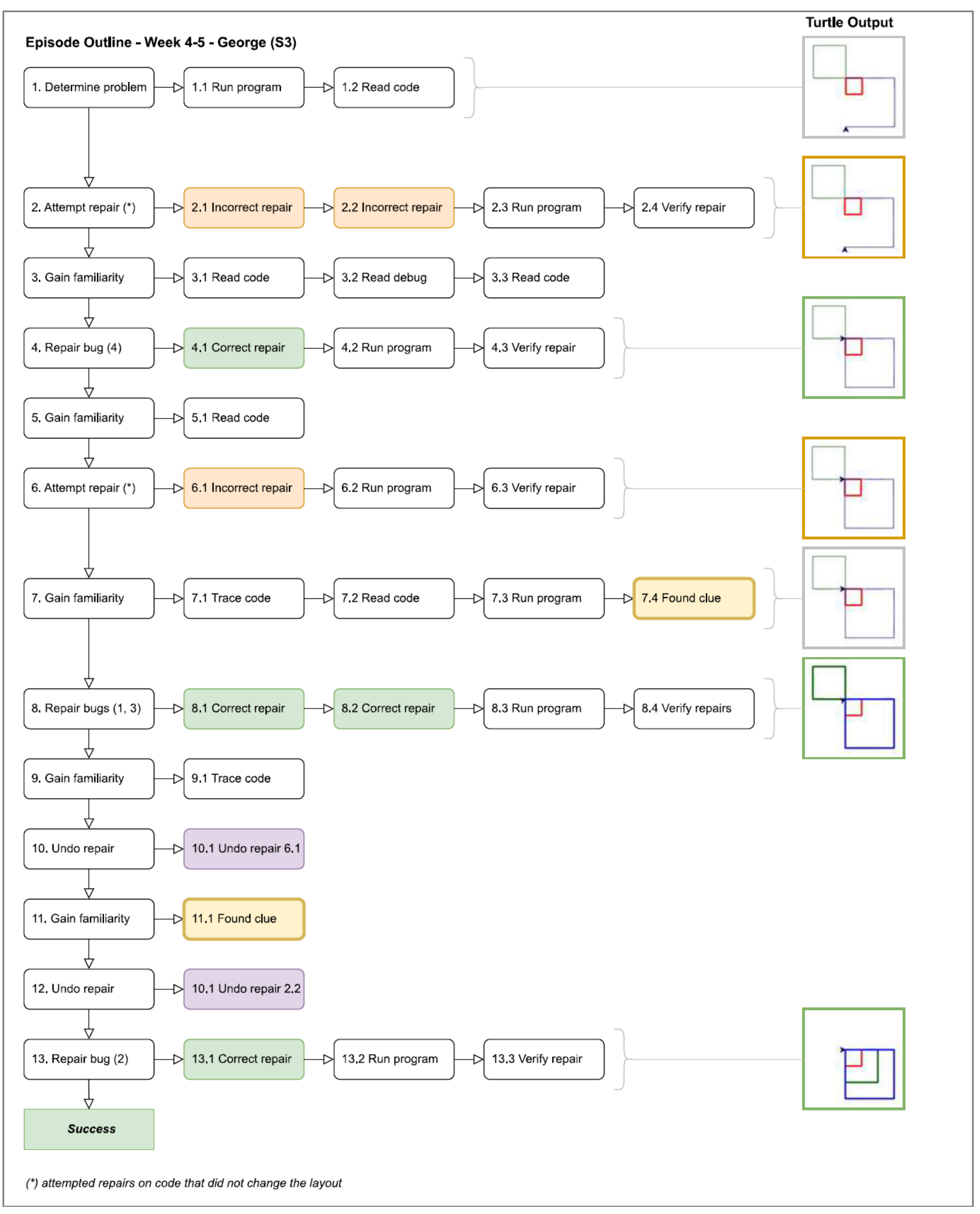


Figure D4. Week 4/5 – George's debugging session episodes.

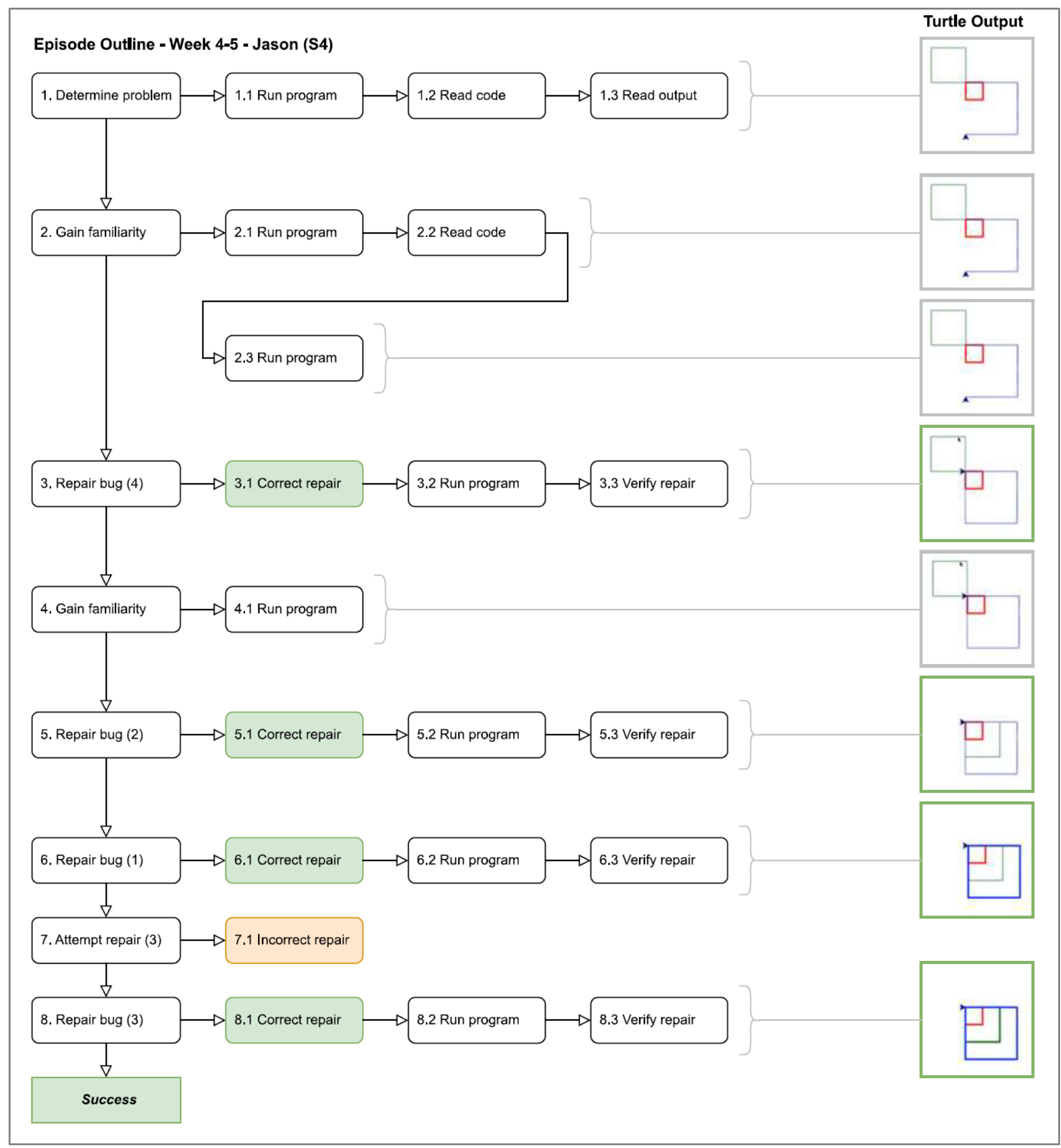


Figure D5. Week 4/5 – Jason's debugging session episodes.

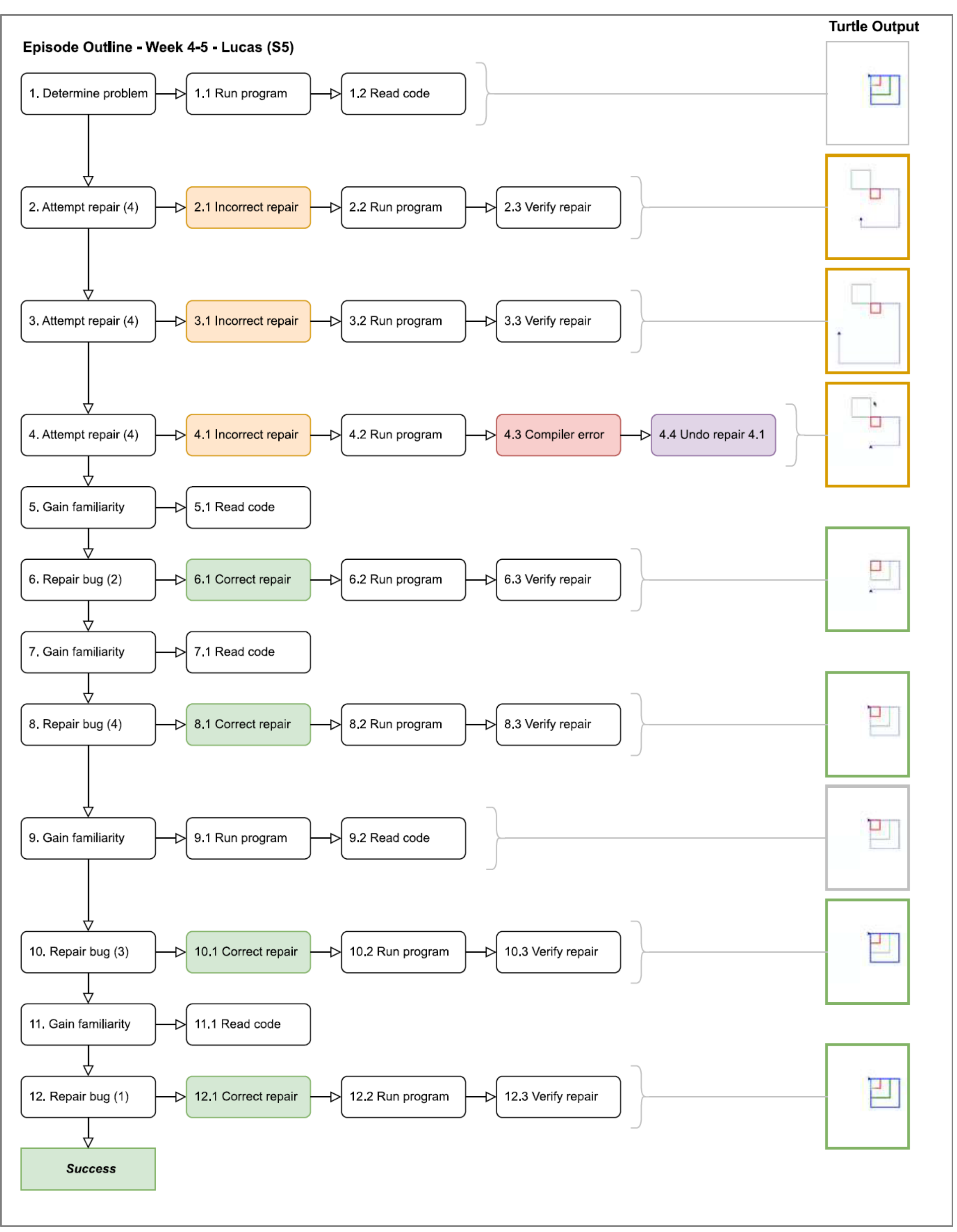


Figure D6. Week 4/5 – Lucas' debugging session episodes.

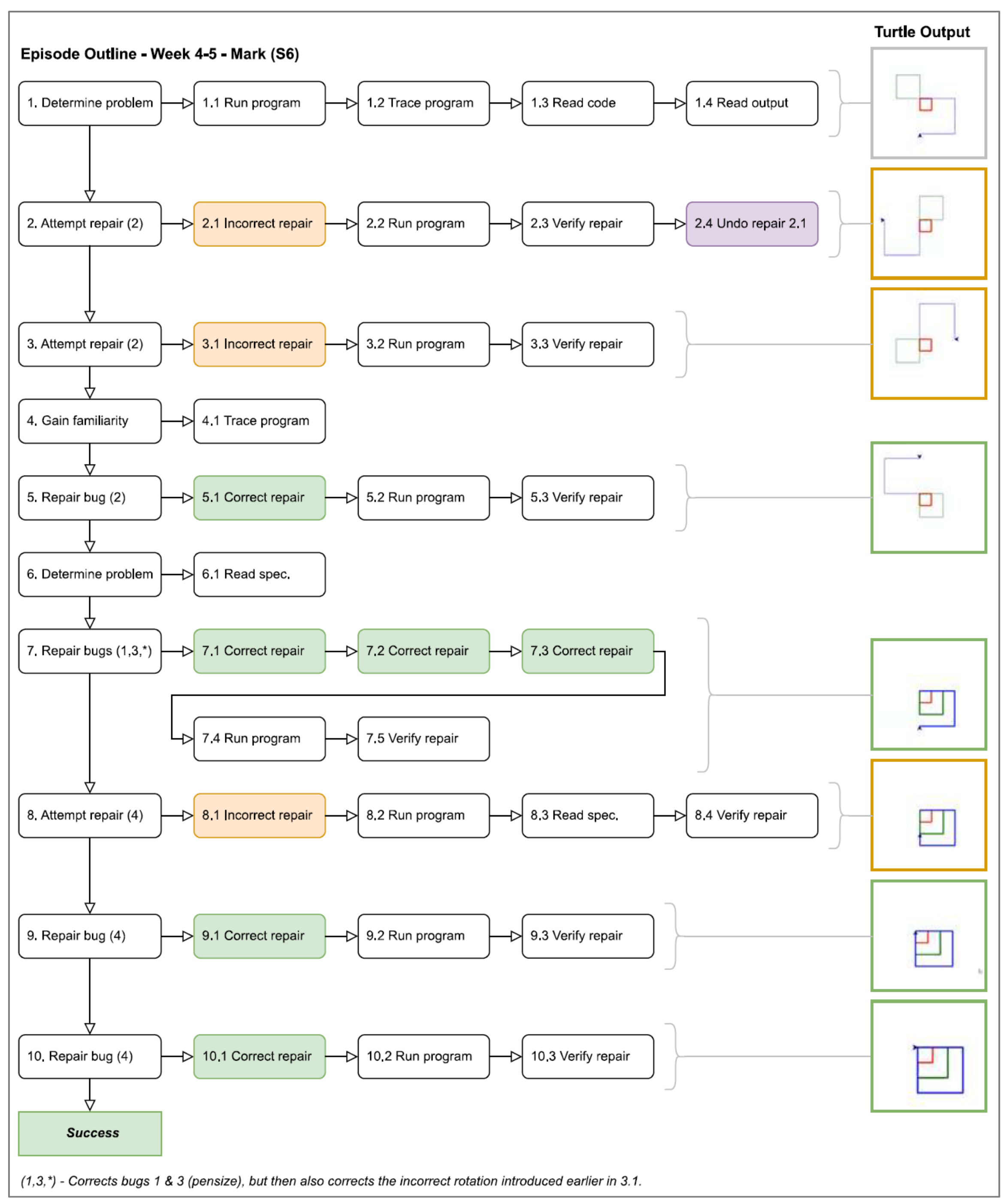


Figure D7. **Week 4/5 – Mark's debugging session episodes.**

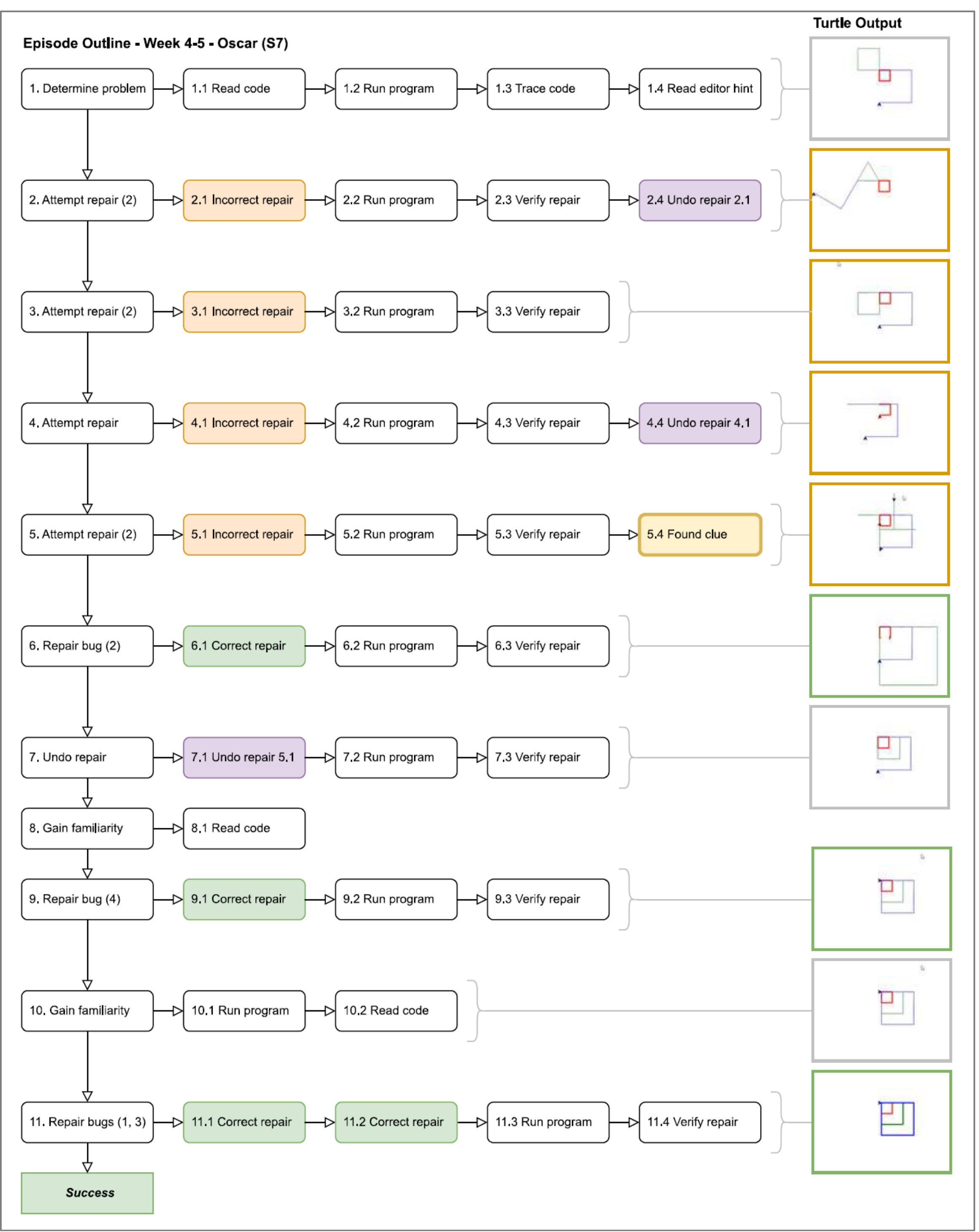


Figure D8. Week 4/5 – Oscar's debugging session episodes.

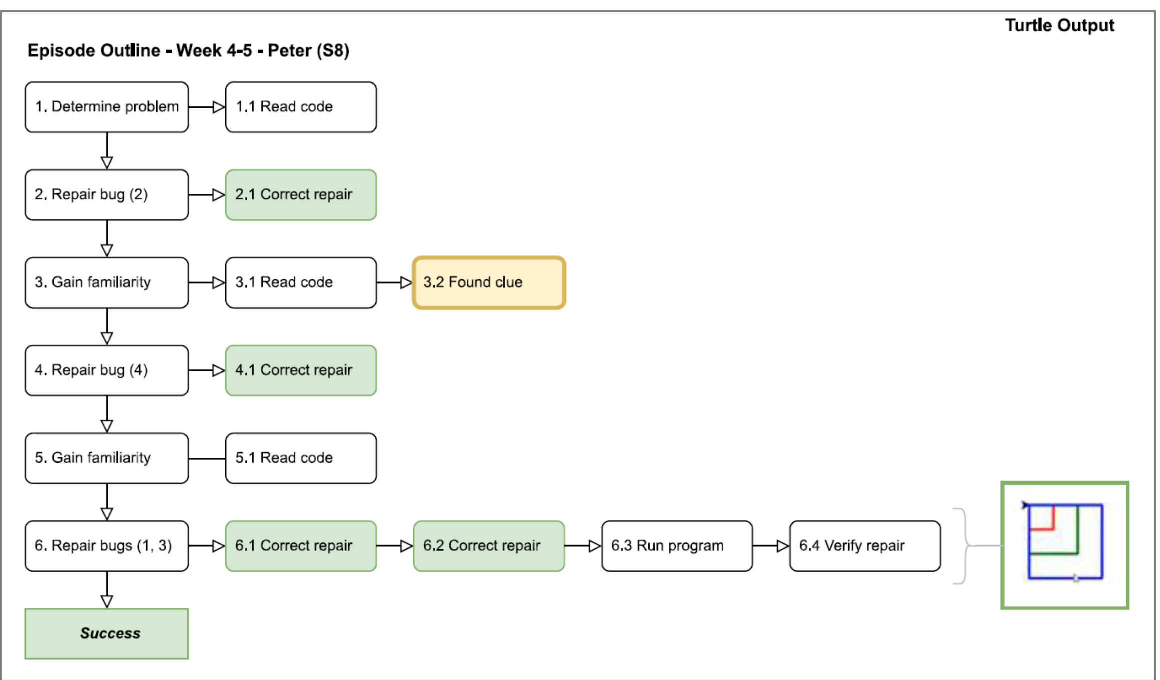


Figure D9. **Week 4/5 – Peter's debugging session episodes.**

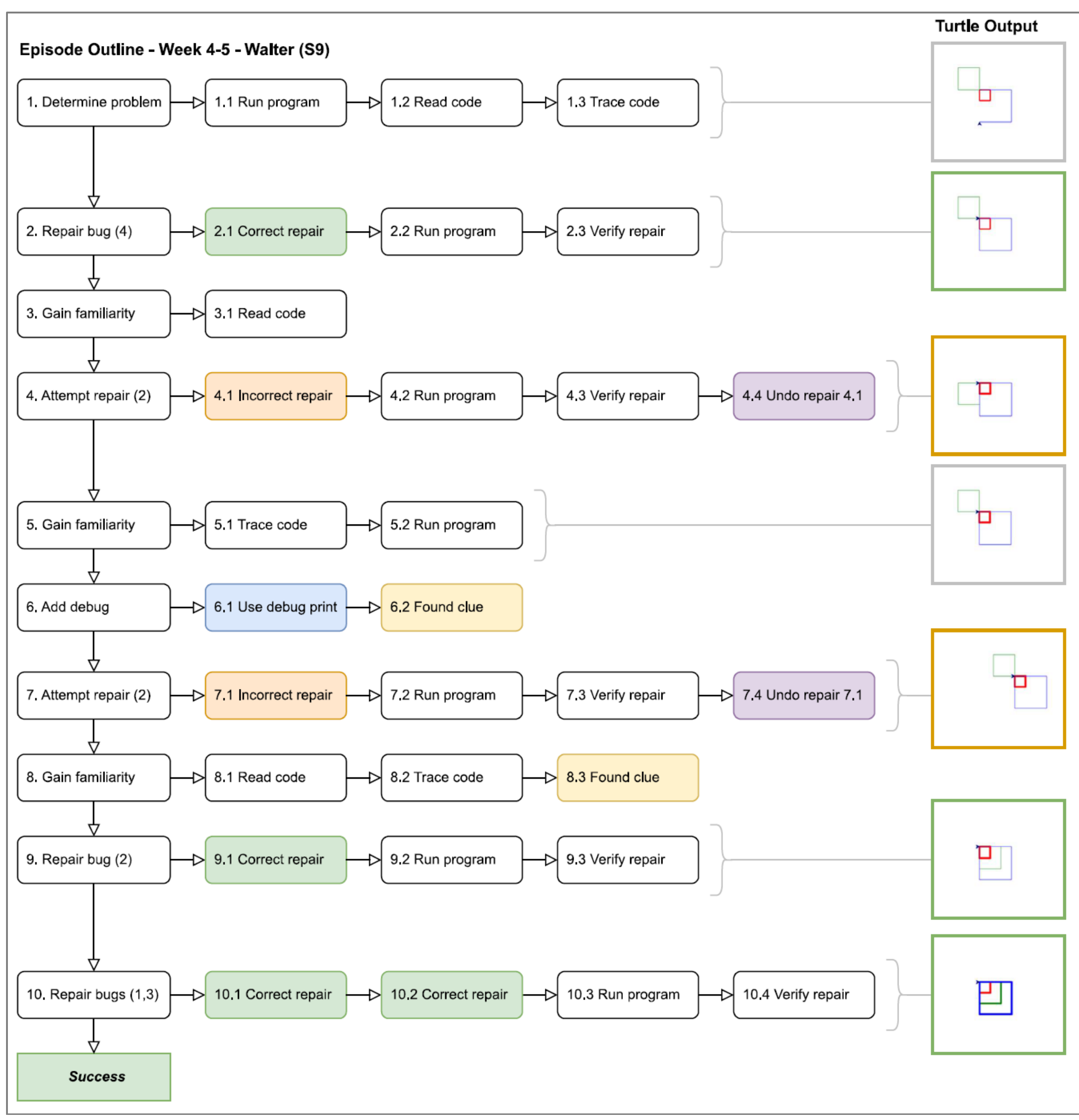


Figure D10. **Week 4/5 – Walter's debugging session episodes.**

Weeks 7/8

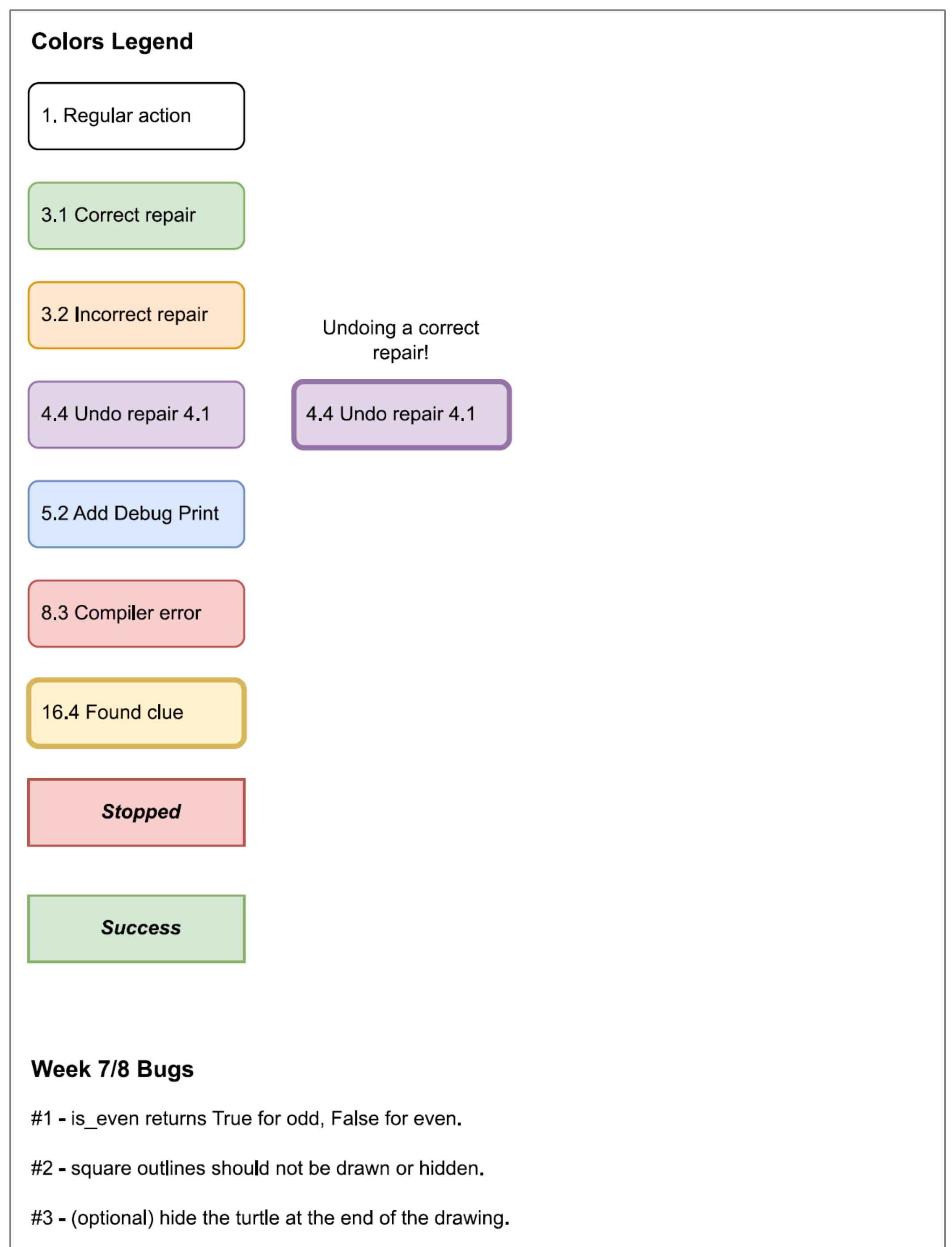


Figure D11. Week 7/8 Debugging Session Episode Colors Legend and Bugs List

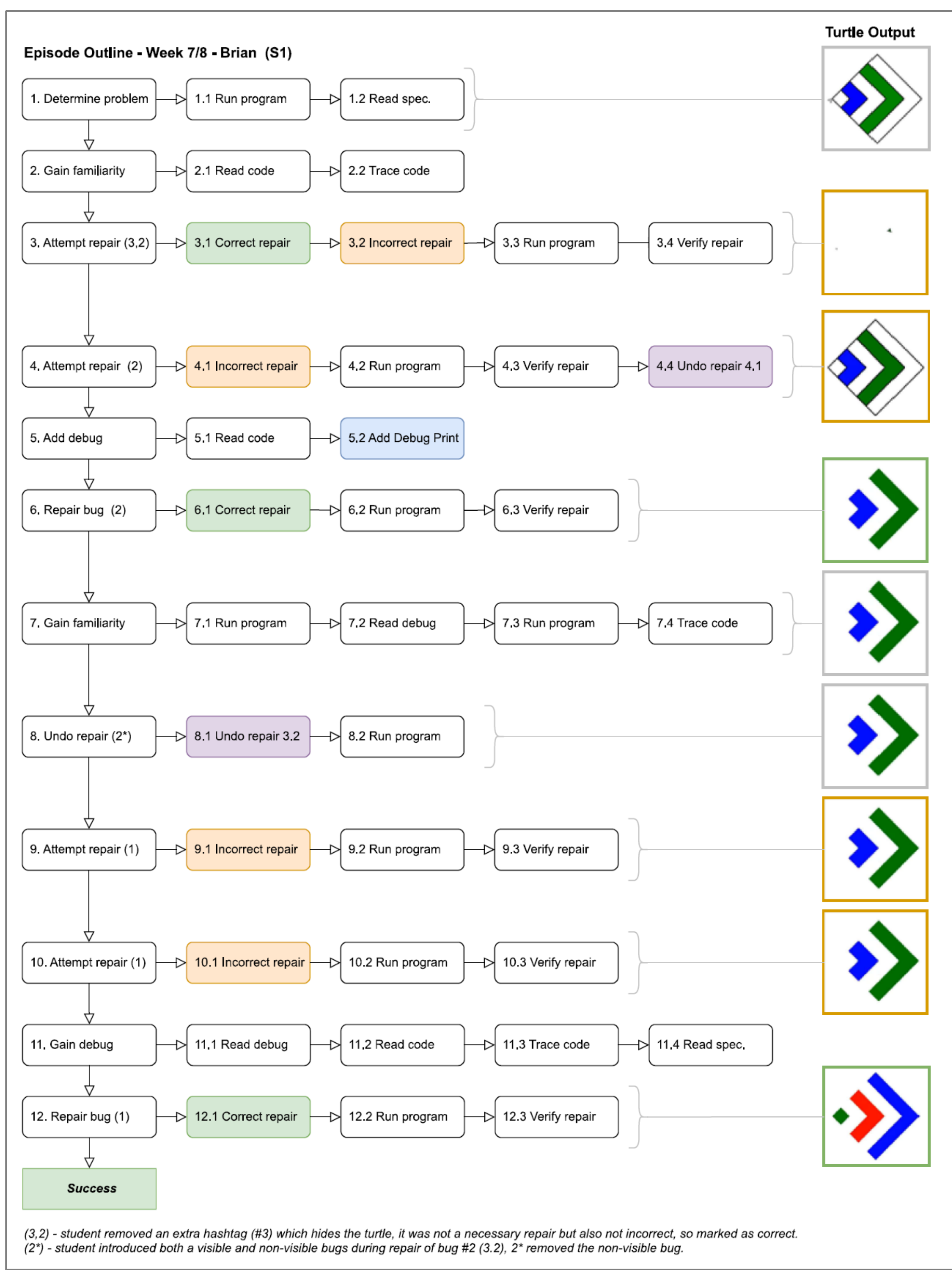


Figure D12. Week 7/8 – Brian's debugging session episodes.

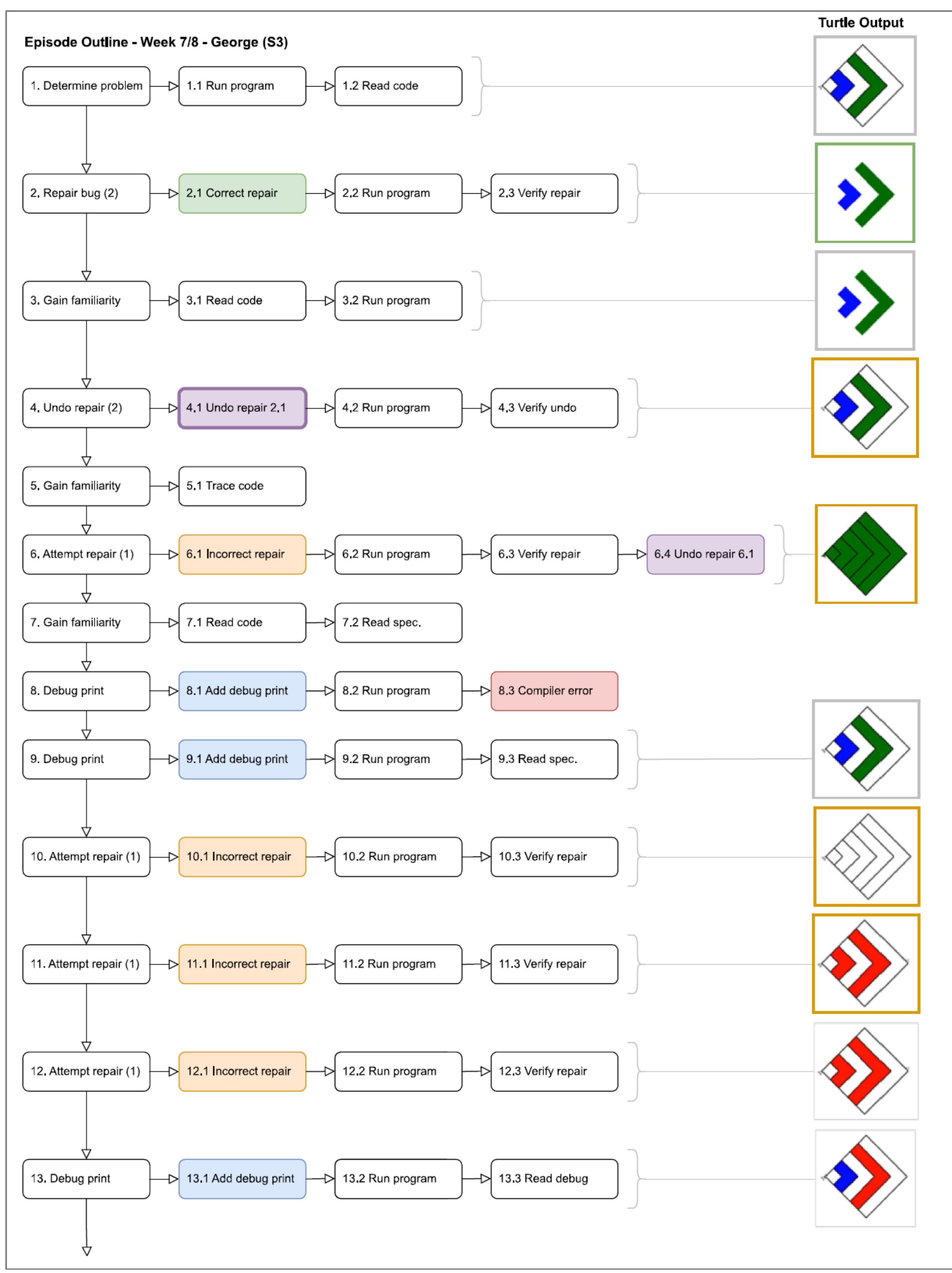


Figure D13a. Week 7/8 – George's debugging session episodes (1 of 2).

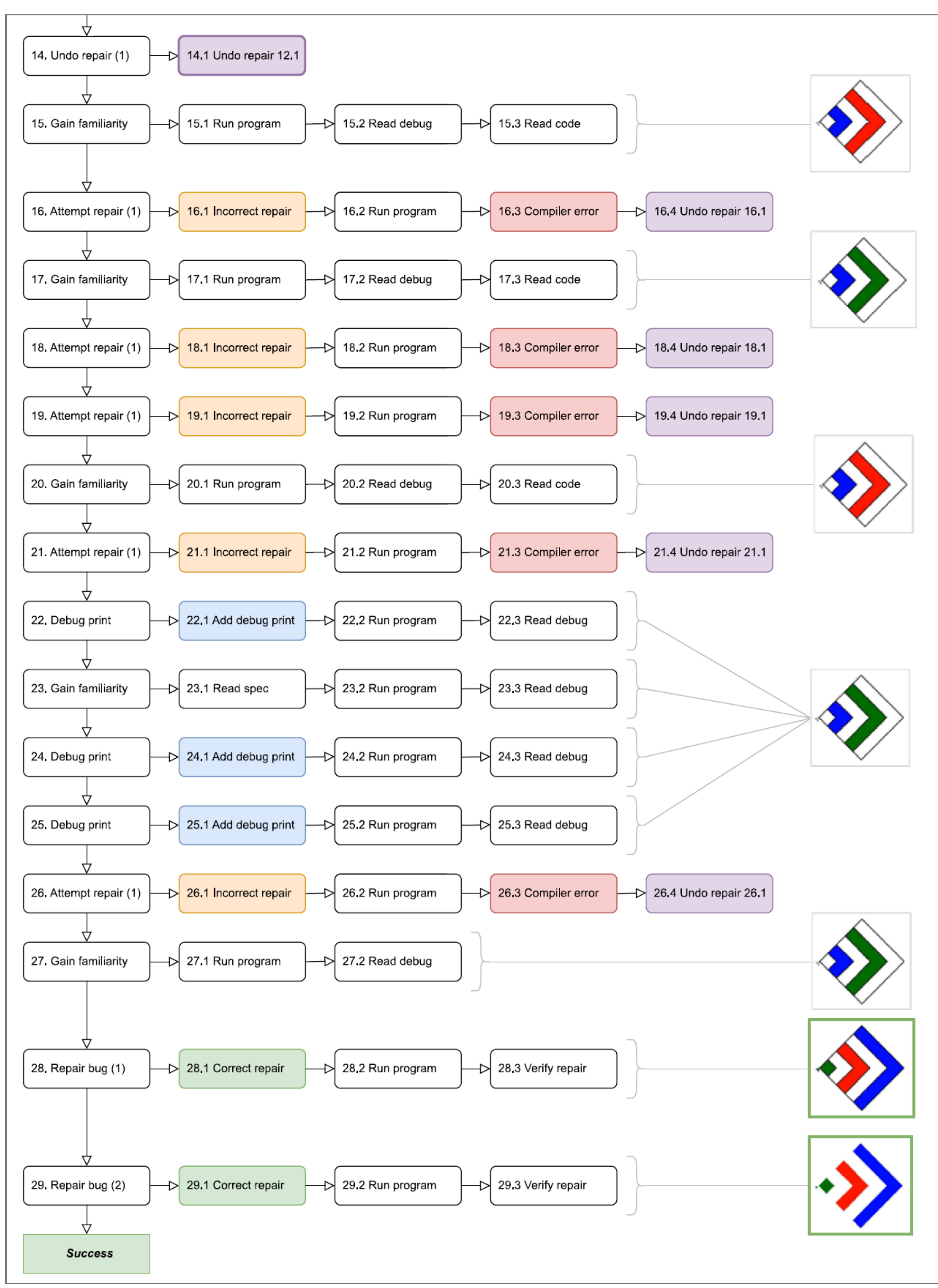


Figure D13b. Week 7/8 – George's debugging session episodes (2 of 2).

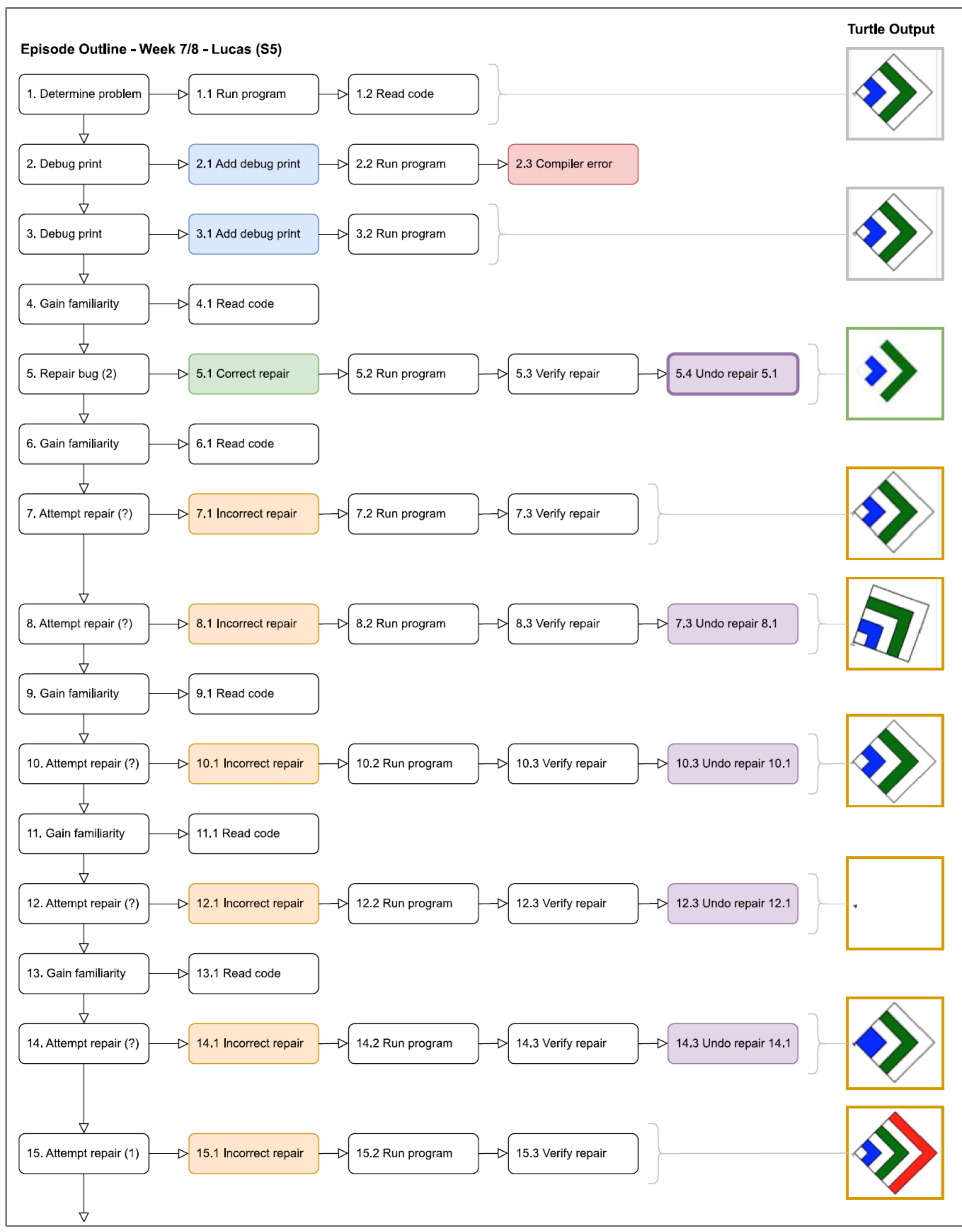


Figure D14a. Week 7/8 – Lucas' debugging session episodes (1 of 2).

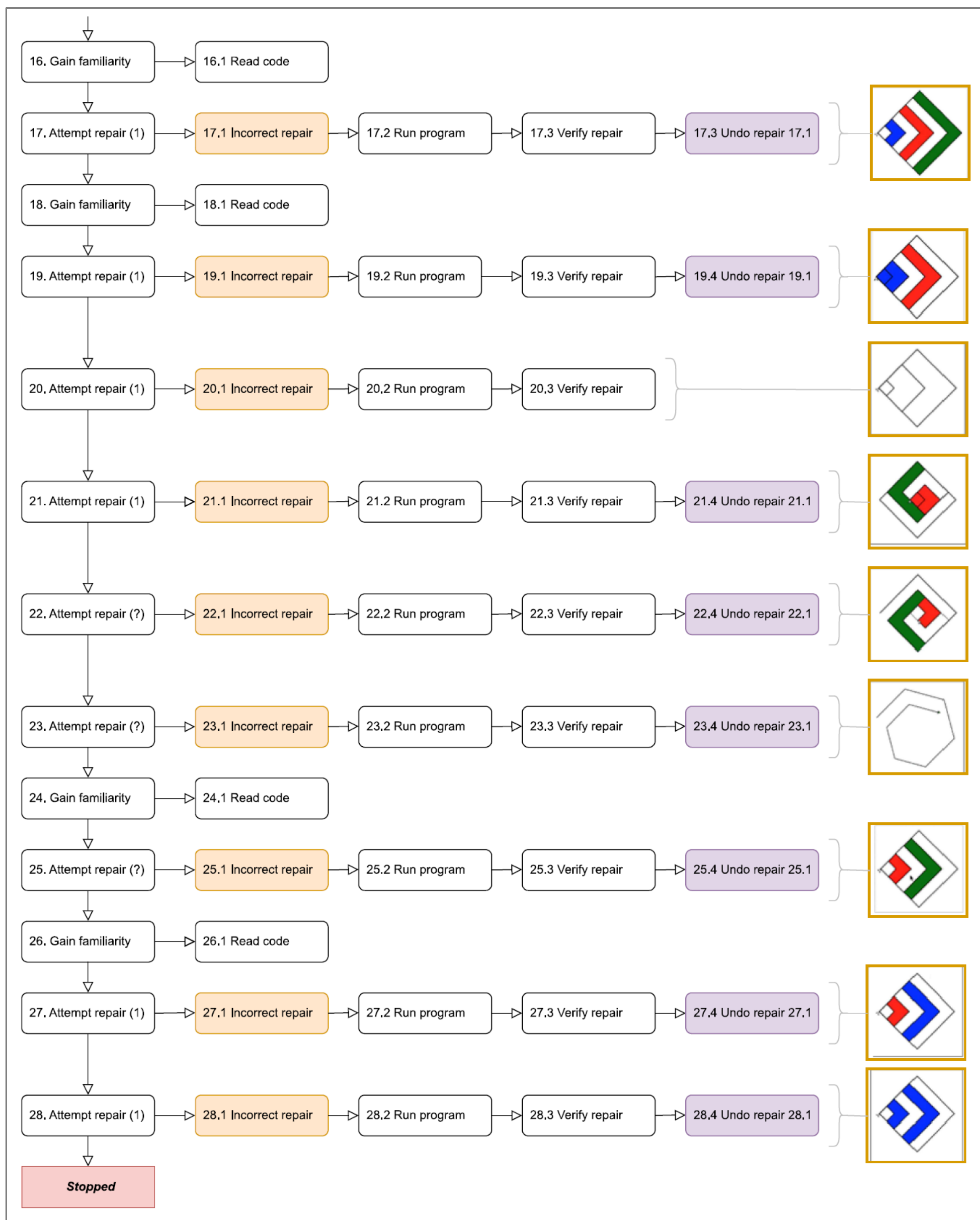


Figure D14b. Week 7/8 – Lucas' debugging session episodes (2 of 2).

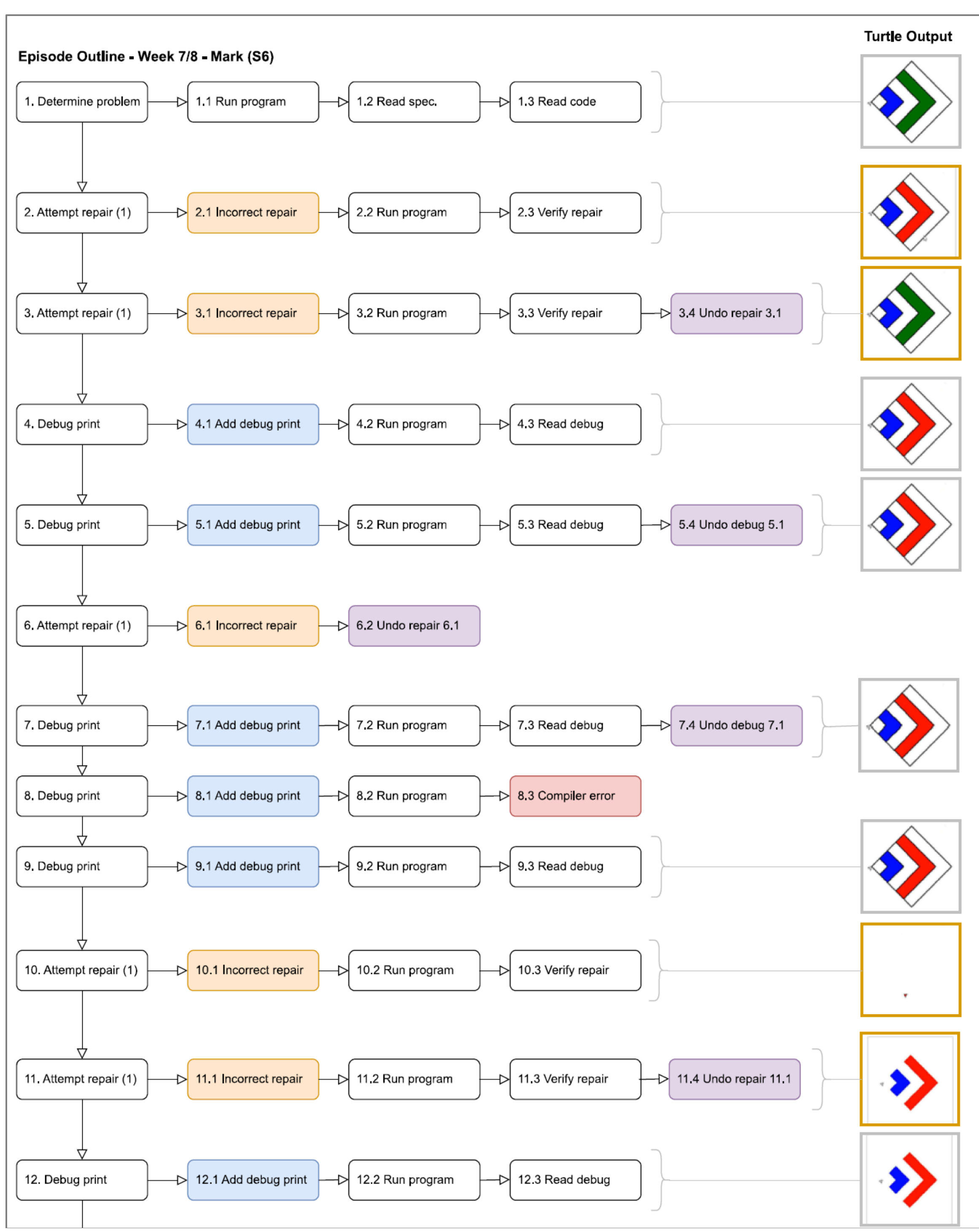


Figure D15a. Week 7/8 – Mark's debugging session episodes (1 of 2).

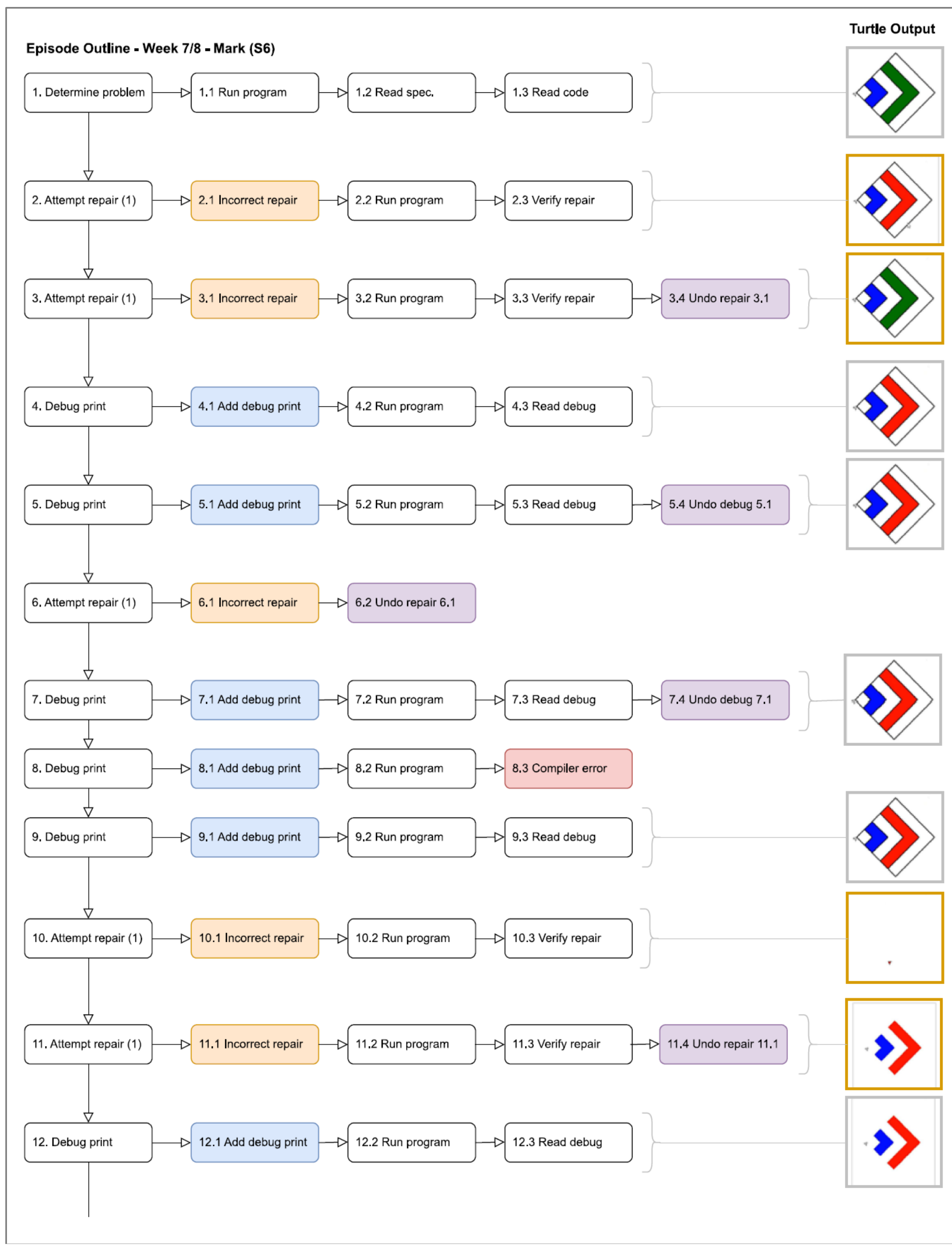


Figure D15b. Week 7/8 – Mark's debugging session episodes (2 of 2).

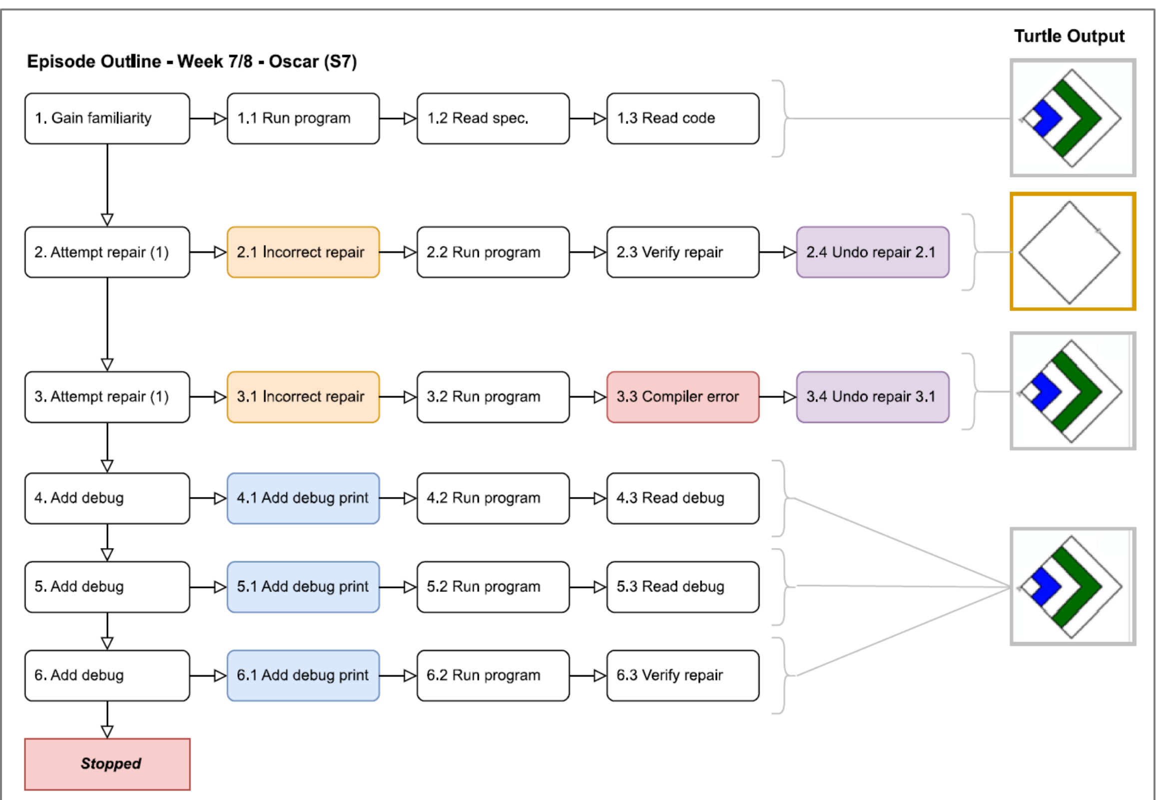


Figure D16. Week 7/8 – Oscar's debugging session episodes.

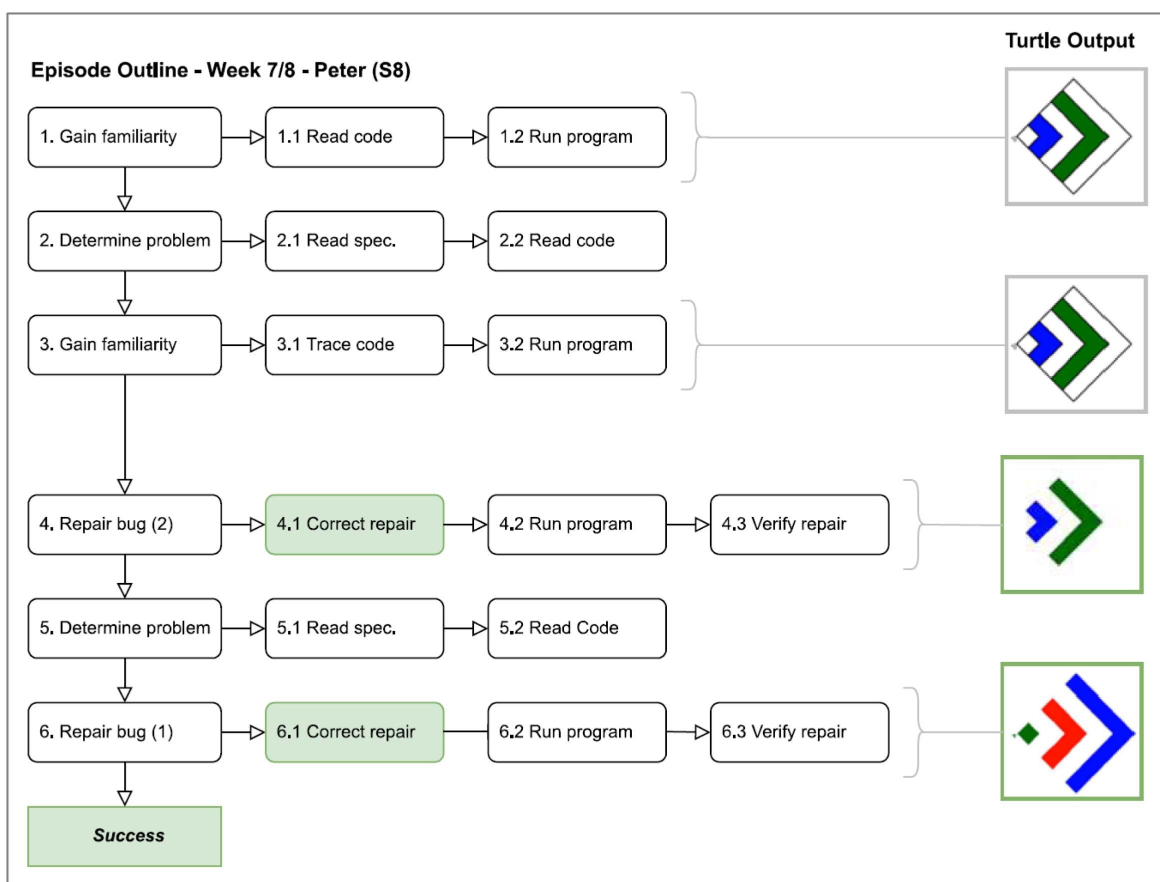


Figure D17. Week 7/8 – Peter's debugging session episodes.

Appendix E.

Recruitment Web Page

SFU Research Study (Fall 2022)

2022-10-03

RE: Research study parental consent request.

Name of study: The code is lava: Improving children's debugging skills with explicit instruction.

SFU ORE Protocol: 30001083

Study Dates: Wednesdays 3:30 PM - 5:30 PM: Oct 19, 26; Nov 2, 9, 16, 23, 30; Dec 7

Cost: None, free

Dear Parent or Guardian,

My name is Chris Kerslake, and for the past 7 years I have been teaching computer science and computer programming to students in Vancouver via after-school programs. During this time, I have observed a lack of student know-how and even an avoidance when it comes to debugging or fixing their code. As a result, I am undertaking a research project as part of my graduate studies at Simon Fraser University (SFU) to look into how students approach debugging and whether teaching them a step-by-step debugging process is beneficial. This letter provides details about the study should you and your child be interested in participating in this study.

Research Project Structure:
This research project will consist of an 8-week Python programming course where students will learn how to create 2D graphics and images. The individual sessions will be 2-hours in length, will be conducted online via Zoom, and will be provided at no cost. During each session, students will first learn and practice a programming technique and then will be given exercises related to programming and debugging to complete. During some of these debugging exercises, students will be asked to join a breakout room individually or in pairs to describe a given problem in detail. In these breakout rooms students will be asked questions about the problem and then asked to talk through how they are trying to solve the problem, a process called thinking aloud. These lessons and breakout sessions will be recorded so that I can analyze them for patterns as part of the research study.

Student Protection and Privacy:
Student protection and privacy are paramount during this study. As a result, students can withdraw from the study at any time and for any reason. As well, all collected information will be anonymized and the video recordings will be transcribed and then destroyed at the end of the research project. Further, student names will not be used in any reporting and all research data will be stored securely and will only be available to Chris Kerslake and his supervisor, Dr. Kevin O'Neill.

How to join the study:
If you wish to participate in the study and agree to allow us to collect and use the information described, please complete both the attached Parent Guardian Informed Consent for Minors form and Student Informed Assent for Minors form and return them to Chris Kerslake.

Thank-you for taking the time to read this letter. We appreciate your time and cooperation.

Contact Details:

Chris Kerslake
Graduate Student
Faculty of Education
Simon Fraser University

Dr. Kevin O'Neill
Associate Professor
Faculty of Education
Simon Fraser University

Appendix F.

Parent Consent form

FACULTY OF EDUCATION
8888 University Drive
Burnaby B.C. Canada V5A 1S6

TEL + 1 778 782 4129
SFU.CA

PARENT/GUARDIAN INFORMED CONSENT FOR MINORS

CONSENT BY PARENT OR GUARDIAN TO ALLOW PARTICIPATION IN A RESEARCH STUDY

Title of Study: The code is lava: Improving children's debugging skills with explicit instruction.

Investigator Name: Chris Kerslake, Student Lead

Supervisor Name: Dr. Kevin O'Neill, Associate Professor

Investigator Department: Faculty of Education

SFU ORE Protocol: 30001083

Simon Fraser University (the University) and those conducting this study subscribe to the ethical conduct of research and to the protection at all times of the interests, comfort, and safety of participants. This form and the attached letter from Chris Kerslake have been given to you for your own protection, and to ensure that you fully understand the procedures, risks, and benefits associated with this study. This document seeks your consent to allow your minor child to participate in the research study described below.

Procedures:

This research study will consist of an 8-week Python programming course where students will learn how to create 2D graphics and images using code. Each weekly session will be 2-hours in length and will consist of instruction, exercises, and interviews. A key part of this study involves the introduction of a formal debugging process to try and help students debug their code more effectively. During some of the sessions, students will be interviewed individually or in pairs and asked to describe their thought processes while trying to debug Python code containing errors. These sessions will be recorded to capture student responses and actions. Following the course, Chris will analyze and report on the data as part of his investigation into procedures aimed at improving the debugging skills of children learning to program with Python.

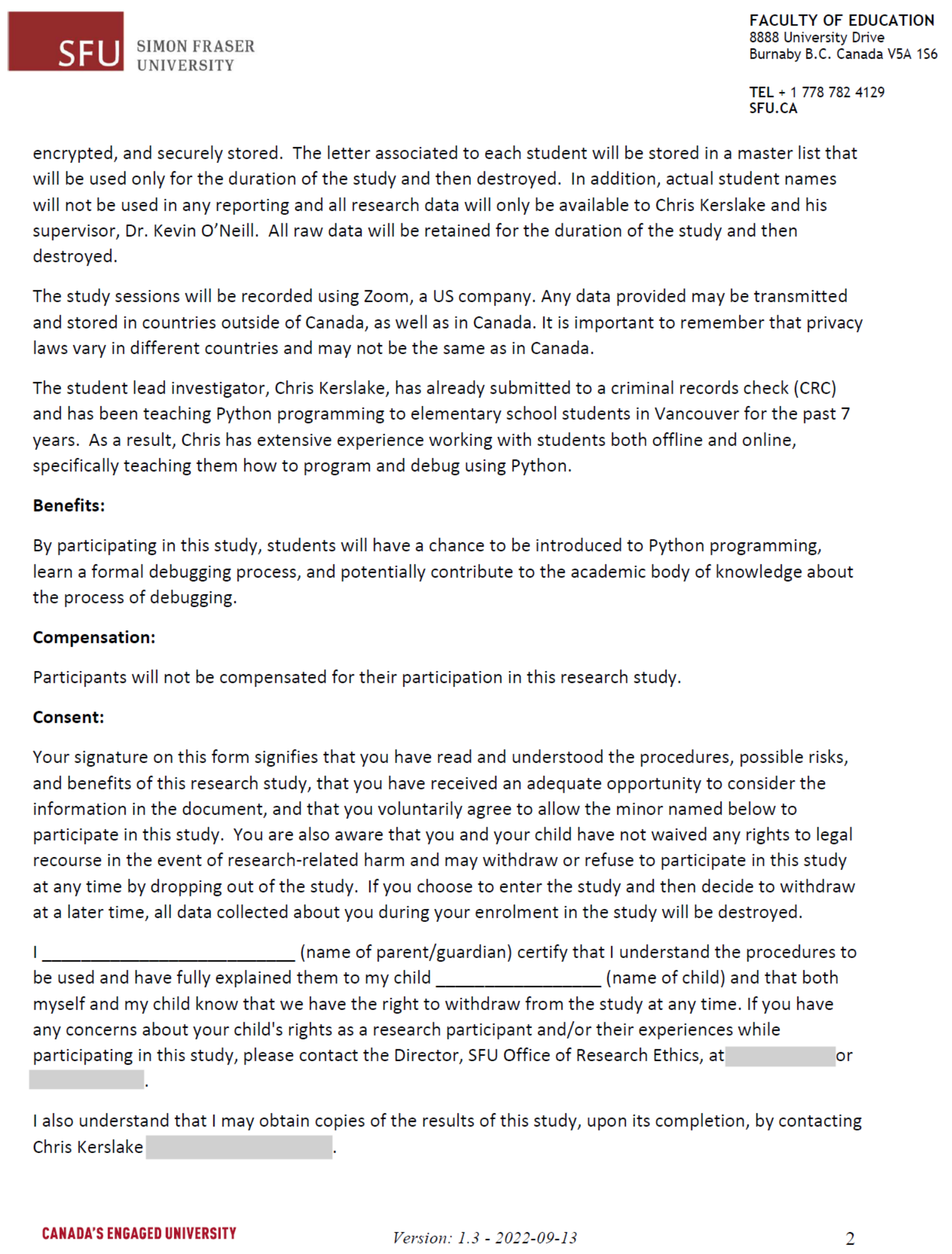
Risks:

This study aims to investigate participants' pre-existing troubleshooting and debugging skills during the normal course of instruction. The circumstances of this study do not introduce any special risk beyond that which a student would encounter in a typical online class. Students may find the process of debugging frustrating but will be supported and guided through the process when necessary.

Students' responses and actions will be recorded for analysis. At the start of the study, each student will be randomly assigned a letter from the alphabet in place of their name. During analysis, all data will be anonymized using these individually-assigned letters, stripped of personally identifiable information,

CANADA'S ENGAGED UNIVERSITY *Version: 1.3 - 2022-09-13* 1

Figure F1. Parent/Guardian Informed Consent for Minors form (page 1 of 3).



SFU SIMON FRASER UNIVERSITY

FACULTY OF EDUCATION
8888 University Drive
Burnaby B.C. Canada V5A 1S6

TEL + 1 778 782 4129
SFU.CA

encrypted, and securely stored. The letter associated to each student will be stored in a master list that will be used only for the duration of the study and then destroyed. In addition, actual student names will not be used in any reporting and all research data will only be available to Chris Kerslake and his supervisor, Dr. Kevin O'Neill. All raw data will be retained for the duration of the study and then destroyed.

The study sessions will be recorded using Zoom, a US company. Any data provided may be transmitted and stored in countries outside of Canada, as well as in Canada. It is important to remember that privacy laws vary in different countries and may not be the same as in Canada.

The student lead investigator, Chris Kerslake, has already submitted to a criminal records check (CRC) and has been teaching Python programming to elementary school students in Vancouver for the past 7 years. As a result, Chris has extensive experience working with students both offline and online, specifically teaching them how to program and debug using Python.

Benefits:

By participating in this study, students will have a chance to be introduced to Python programming, learn a formal debugging process, and potentially contribute to the academic body of knowledge about the process of debugging.

Compensation:

Participants will not be compensated for their participation in this research study.

Consent:

Your signature on this form signifies that you have read and understood the procedures, possible risks, and benefits of this research study, that you have received an adequate opportunity to consider the information in the document, and that you voluntarily agree to allow the minor named below to participate in this study. You are also aware that you and your child have not waived any rights to legal recourse in the event of research-related harm and may withdraw or refuse to participate in this study at any time by dropping out of the study. If you choose to enter the study and then decide to withdraw at a later time, all data collected about you during your enrolment in the study will be destroyed.

I ___________________________ (name of parent/guardian) certify that I understand the procedures to be used and have fully explained them to my child _________________ (name of child) and that both myself and my child know that we have the right to withdraw from the study at any time. If you have any concerns about your child's rights as a research participant and/or their experiences while participating in this study, please contact the Director, SFU Office of Research Ethics, at [redacted] or [redacted].

I also understand that I may obtain copies of the results of this study, upon its completion, by contacting Chris Kerslake [redacted].

CANADA'S ENGAGED UNIVERSITY   *Version: 1.3 - 2022-09-13*   2

Figure F2. Parent/Guardian Informed Consent for Minors form (page 2 of 3).

SFU SIMON FRASER UNIVERSITY

FACULTY OF EDUCATION
8888 University Drive
Burnaby B.C. Canada V5A 1S6

TEL + 1 778 782 4129
SFU.CA

Child Participant:

Name of child: (print) ____

Parent or Guardian:

Name of Parent, Guardian, or other (please print): ____

Relationship to the child: (please print): ____

Parent/Guardian Signature: ____

Date: ______________________

CANADA'S ENGAGED UNIVERSITY

Version: 1.3 - 2022-09-13

3

Figure F3. Parent/Guardian Informed Consent for Minors form (page 3 of 3).

Appendix G.

Student Assent form

FACULTY OF EDUCATION
8888 University Drive
Burnaby B.C. Canada V5A 1S6

TEL + 1 778 782 4129
SFU.CA

STUDENT INFORMED ASSENT FOR MINORS

ASSENT FROM MINOR PARTICPANTS TO PARTICIPATE IN A RESEARCH STUDY

Title of Study: The code is lava: Improving children's debugging skills with explicit instruction.

Investigator Name: Chris Kerslake, Student Lead

Supervisor Name: Dr. Kevin O'Neill, Associate Professor

Investigator Department: Faculty of Education

SFU ORE Protocol: 30001083

My name is Chris Kerslake, and I am a researcher at Simon Fraser University (also called SFU). I am inviting you to take part in my research study. In my study, I am trying to learn how kids aged 10-12 learn how to fix broken code while programming, something we call debugging. When programming, kids spend a lot of time fixing code, so I want to learn more about how they actually do it. This letter has been given to you to tell you about the study, see if you are interested, and if so, ensure you understand your rights during the study.

It is your choice if you want to take part in the study.

It is completely your choice if you want to take part in the study, and you can stop at any time if you decide that you no longer want to be in the study. Also, if you change your mind about being in the study, you can stop at any time by telling your parents/guardian or me. If you do decide not to be in the study or decide to stop participating in the study, no one will be upset with you, and you will not get in any trouble.

What will happen if you decide to be in the study?

If you decide that you want to be in the study, you will attend eight online Zoom session where I will teach you how to write Python code to create 2D graphics and images. During these sessions, I will ask you to try and debug some broken code and talk about what you see and how hard the problem was. You can work on the problems on your own or with a partner. With your permissions, I will record these sessions so I can see how students solve debugging problems. At the end of the study, I will ask for your feedback on the study and your thoughts on debugging. After the study, I will review the recordings and write a report on how students debug their code.

Are there any risks when taking part in this study?

I do not believe that being in this study will harm you in any way, but there is a chance that you might find some of the questions frustrating to figure out. You do not have to answer any of the questions if you do not want to, and you can stop taking part in the study at any time.

CANADA'S ENGAGED UNIVERSITY   *Version: 1.3 - 2022-09-13*   1

Figure G1. Student Informed Assent for Minors form (page 1 of 2).

FACULTY OF EDUCATION
8888 University Drive
Burnaby B.C. Canada V5A 1S6

TEL + 1 778 782 4129
SFU.CA

Who will know you were in the study?

Your parents or guardian know that you might be in the study, but I will not tell anyone else. You and the other kids in the study will be able to see your name in Zoom, see what you do, and hear what say, but I will make sure that no one else knows what you said, or which answers you gave me.

Please circle yes or no:

Has somebody explained this study to you?	Yes	No
Do you understand what the study is about?	Yes	No
Have you asked all the questions you want?	Yes	No
Have you had your questions answered in a way you understand?	Yes	No
Do you understand it is OK to stop at any time?	Yes	No
Do you want to be in the study?	Yes	No
Is it OK if I record the interviews?	Yes	No

If you want to take part in this study, please print or write your name and today's date.

Your Name: ___________________________________

Today's Date: __________________________________

Your parent or guardian must write their name here too if they are happy for you to do the study.

Parent/Guardian Full Name: ___________________________________

Parent/Guardian Signature: ___________________________________

Today's Date: __________________________________

CANADA'S ENGAGED UNIVERSITY *Version: 1.3 - 2022-09-13* 2

Figure G2. Student Informed Assent for Minors form (page 2 of 2).