June, 2023

A Think-Aloud Study of Novice Debugging

Jacqueline Whalley, *Auckland University of Technology*
Amber Settle
Andrew Luxton-Reilly, *University of Auckland*

A Think-Aloud Study of Novice Debugging

JACQUELINE WHALLEY, Auckland University of Technology, New Zealand
AMBER SETTLE, DePaul University, USA
ANDREW LUXTON-REILLY, University of Auckland, New Zealand

Debugging is a core skill required by programmers, yet we know little about how to effectively teach the process of debugging. The challenges of learning debugging are compounded for novices who lack experience and are still learning the tools they need to program effectively. In this work, we report a case study in which we used a think-aloud protocol to gain insight into the behaviour of three students engaged in debugging tasks. Our qualitative analysis reveals a variety of helpful practices and barriers that limit the effectiveness of debugging. We observe that comprehension, evidence-based activities, and workflow practices all contribute to novice debugging success. Lack of sustained effort, precision, and methodical processes negatively impact debugging effectiveness. We anticipate that understanding how students engage in debugging tasks will aid future work to address ineffective behaviours and promote effective debugging activities.

CCS Concepts: • **Social and professional topics → Computing education;**

Additional Key Words and Phrases: Debugging, introductory programming, think-aloud

ACM Reference format:
Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2023. A Think-Aloud Study of Novice Debugging. *ACM Trans. Comput. Educ.* 23, 2, Article 28 (June 2023), 38 pages.
https://doi.org/10.1145/3589004

1 INTRODUCTION

Debugging programs has been identified as an essential programming skill [24, 77]. Unfortunately, despite the continued presence of debugging in the Software Development Fundamentals of the ACM curriculum [30, 38], it remains a neglected part of computing education practice. After decades of literature calling for approaches to teach debugging [16, 17, 40, 46, 70], we still have no clear guidance on what to teach and how to teach it [71]. A part of the issue is that debugging programs is a complex cognitive process [8]. When a program has a bug, further development is prevented until the bug is fixed. This means that poor debugging practices can block students from progressing in their learning.

There have been decades of studies investigating how students learn to debug [41, 70, 72], including multiple think-aloud studies examining student debugging [28, 55, 72, 94]. However, despite extensive work on understanding student debugging, there are few detailed, qualitative studies

Authors' addresses: J. Whalley, Department of Computer Science and Software Engineering, Auckland University of Technology, 55 Wellesley Street East, Auckland CBD, Auckland 1010, New Zealand; email: jacqueline.whalley@aut.ac.nz; A. Settle, School of Computing, DePaul University, 243 S. Wabash Avenue, Chicago, Illinois, USA; email: asettle@cdm.depaul.edu; A. Luxton-Reilly, School of Computer Science, University of Auckland, 38 Princes St, Auckland CBD, Auckland 1010, New Zealand; email: a.luxton-reilly@auckland.ac.nz.

1946-6226/2023/06-ART28 $15.00
https://doi.org/10.1145/3589004

of the debugging practices of novice programmers. Many of the previous studies were conducted prior to the adoption of now-common programming languages and development environments. Our exploratory study uses Python and the IDLE development environment, a context that has become increasingly common in higher education [82]. Much of the previous work on understanding student errors justifiably focused on compilation errors since they prevent students from obtaining any information about their programs [6, 22, 64]. As an interpreted language, Python provides a different context for students, and the errors are not always analogous with other commonly taught languages such as Java [74]. In this work, we investigate student debugging using several frameworks found in the debugging literature, including various approaches to the debugging process [21, 39, 53, 66, 95] and ways of classifying debugging behavior [66].

Our study of three students adopts a think-aloud protocol to understand how students debug in a first-year programming course taught in Python. Students in the course were taught a debugging process, provided with several worked examples, and then asked to complete paper-and-pencil debugging lab activities. Several weeks later, students were invited to participate in a think-aloud debugging session not associated with the class. Our aim was to observe students debugging in an unstructured manner without external direction.

The activities that participants were given to debug varied from a simple off-by-one looping exercise used as a warm-up to a complex multi-function implementation of the rainfall problem with several bugs. The bugs injected into the programs were discussed in previous literature on novice student errors [4, 5, 13, 31, 36, 42, 61, 63, 83, 88] and commonly seen by us in the classroom. The participants were provided with buggy code and with several test cases demonstrating the bugs present. There was no direct guidance or suggestions provided to the students about how to approach the debugging process beyond the in-class lab activity.

Our goal was to observe and understand effective and ineffective debugging practices employed by students who were relatively inexperienced in both programming and debugging. In this article, we use the term *novice* to refer to students in their first year of study in tertiary programming courses. Our research questions are as follows:

RQ1: What novice debugging practices are effective?
RQ2: What novice debugging practices are ineffective?

By understanding the processes and practices that students default to when debugging in an unstructured activity, we hope to gain understanding about the alignment of the practices with existing frameworks for debugging. As a result of this work, we summarise several effective practices used by students and also observe a lack of precision, failure to follow through, and an inability to compensate for nonexistent or fragile knowledge, which contributes to inefficient debugging. We conclude by offering several suggestions for teaching debugging.

The main contributions of this study are providing rich and detailed accounts of three novice students engaged in debugging practices (of which there are few other similar case studies), providing evidence of both effective and ineffective practices, suggesting teaching activities to develop effective novice programmer processes, and providing a set of themes for future analysis of novice debuggers.

2 RELATED LITERATURE

2.1 Knowledge and Process

An early review of debugging literature by Ducasse and Emde [25] identified seven different kinds of knowledge required to locate bugs, four of which relate to program comprehension. A subsequent review of debugging literature by McCauley et al. [65] confirmed the importance of program comprehension in the debugging process.

Schaafstal et al. [79] describe debugging as a specialised version of troubleshooting. In one framework for troubleshooting, Jonassen and Hung [39] distinguish between *domain knowledge*, which would include understanding of the syntax and semantics of a programming language, and *system knowledge*. Program comprehension is critically important in the development of the system knowledge that is required to debug that system (i.e., the program). However, although program comprehension is necessary for effective debugging, it is not sufficient [1] — the *process* used to engage in debugging is critically important. A case study documenting a student's natural debugging behaviour found that the student had relevant domain knowledge, but the main factor impacting the processes used was student attention rather than domain knowledge [52].

Zeller's debugging steps, elaborated in [21] and [53], describe an iterative process that involves:

(a) Observing a failure (i.e., a difference between the expected and actual behaviour).
(b) Proposing an explanation for the failure.
    (i) Conducting experiments to test the explanation by collecting more observations.
    (ii) Modifying the explanation until it describes the precise cause of the failure.
(c) Correcting the program and evaluating the correction.

The most complex of these steps is typically (b), in which a programmer develops a hypothesis for the cause of the faulty behaviour. This step typically involves gathering further information. The way a programmer goes about collecting this information and formulating a resultant hypothesis depends on the strategy the programmer employs.

Metzger [66] describes debugging as the process of determining *why* a given set of inputs causes an unacceptable behaviour in a program and *what must be changed* to cause the behaviour to be acceptable — this aligns with step (b). In describing different aspects of the debugging process, Metzger distinguishes between debugging strategies, debugging heuristics, and debugging tactics. These distinctions have utility in viewing debugging though an educational lens, but the term *heuristics* is already used in other computer science contexts and may introduce confusion when used as a category here. To minimise possible confusion, we use the term *scaffolding* as an alternative. This term aligns with the activities described by Metzger [66] as heuristics, but conveys the purpose of the activities in an educational context more appropriately as a bridge between the low-level tactics and high-level strategies. We therefore organize the related work about debugging processes according to the categories of *strategies, scaffolding,* and *tactics*.

2.2 Strategies

Metzger [66] characterizes debugging strategies as general approaches that consist of three parts: a set of assumptions, a control structure, and an evaluation mechanism. The evaluation is a self-reflective process that is critically important, as it provides information about the success of the strategy. This information can be used to decide when a different strategy should be employed. Such strategies include algorithmic approaches such as depth-first or breadth-first searches, program slice analysis, deductive analysis, and inductive analysis.

Strategies are broad, high-level approaches that might be more typically associated with experts, and although novice debugging literature uses the term *strategy*, it is often used to refer to activities that would fall into other categories described by Metzger [66]. However, several studies do refer to broad strategies that are adopted by novice programmers.

Jonassen and Hung [39] identify both forward and backward reasoning as strategies that are deployed for debugging by both experts and novices. Several studies reported that when trying to understand the intended functionality of a program written by others, novices tended to use forward reasoning [28, 41]. Contrasting with these findings, Yen et al. [94] reported that students debugging programs authored by others in the C language tended to use backward reasoning. It is possible that the nature of the bug and the language in use may influence the strategies employed by students.

Eranki and Moudgalya [26] focused on teaching one technique, known as *program slicing*. This strategy involves slicing the program into small code fragments to aid debugging comprehension. Another strategy deployed by successful students was taking the time to consider a variety of alternative causes for the bug rather than simply working on one or two hypotheses [29, 55]. Weighing up the likelihood of different alternative cases and the possible cost of investigating each requires a strategic lens.

2.3 Scaffolding

Scaffolding activities are the rules of thumb that may guide debugging practices. This includes approaches such as building an understanding of the overall problem structure; explaining the problem to someone else; identifying key features of the problem; categorizing the problem (for example, in terms of efficiency, robustness, correctness); using testing to narrow the problem; and stabilizing the problem.

Explaining: Explaining a problem, or how the code works, to another person can improve the understanding of a problem, or reveal misunderstanding about code. Asking students to explain their problem (and/or code) to a tutor can improve debugging by making the student mental model more explicit [62]. The "rubber-duck" debugging approach, the act of verbalizing how the code works, is a scaffolding approach used by professional software developers [86] that has also been adopted by teachers [10]. Liu et al. [55] evaluated self-explanation quality, finding that it was strongly linked to problem-solving ability in a debugging game, and Rezel [76] reported that self-explanations were an effective scaffold for problem-solving.

Causal reasoning: The ability to make connections between the symptoms of a bug and possible causes allows a software developer to narrow the range of code that they consider, which is an important aspect of effective bug location [41]. Causal reasoning may be used in conjunction with *explaining* discussed earlier but shifts the focus to explanations of relationships between components in the system and the generation of hypotheses based on causal reasoning about the symptoms.

2.4 Tactics

Tactics are specific actions that involve the code more directly. These include actions such as reading the source code, adding a statement to print the contents of a variable, using tools present in debuggers to generate additional information, adding assertions to code to make assumptions explicit, and printing the contents of a data structure.

Prior work on debugging shows that several common tactics are typically employed successfully by students.

Print statements: These can be used to support understanding of *flow of control* by providing an external representation of program execution. Statements are often used when entering a function call to trace or "log" the flow of control. Print statements are also used to make the contents of variables visible to the programmer in real time as the program is executed. Several observational studies report that students deploy this tactic successfully [2, 23, 28, 29, 41, 55, 70, 75].

Code tracing: One of the most effective tactics employed by students was code tracing [28, 29, 35, 41, 45, 54]. Code tracing is effective because it allows the programmer to comprehend the step-by-step execution of the code. This helps to identify the impact of mistakes in the code.

Patterns: Jonassen and Hung [39] describe a framework in which prior experience plays an important role in identifying faults. Several studies investigating the debugging behaviour of introductory programming students observed the use of pattern matching in which students changed code that "didn't look right" based on the code that they have previously seen or written [28, 29, 35, 70].

Seeking help: Students of introductory programming were observed seeking additional information to verify or gain understanding of programming concepts. Students used programming language documentation and online resources [23, 29, 70], but consulting formal textbooks was rare [70].

Bug localization: Four studies saw students isolating the code to discover the problematic area through commenting out or altering code [15, 16, 23, 70].

Use of breakpoints: Breakpoints were rarely used or not mentioned [16].

2.5 Ineffective Practices

Students who were learning programming employed several approaches that were ineffective.

Tinkering: Many studies describe novices making changes to the code using a *trial-and-error* approach, that is, changing the code based on intuition rather than a causal chain of reasoning [15, 16, 28, 29, 55, 70]. Such behaviour suggests a lack of understanding of the code or a lack of hypotheses about the cause of the bug.

Printing: Although print statements can be used effectively, Murphy et al. [70] observed students printing uninformative statements (e.g., "hello") at various places in the code that conveyed limited information about flow of control. For example, one student wrote the same print statement in both conditions of an if-else statement, resulting in an ineffective application of a useful tactic.

Patterns: Although experience is critically important in forming "intuition" about a bug, the limited experience of introductory students can hamper debugging when students believe that their code looks "suspicious" and remove correct code [55].

2.6 Think-Aloud Studies

One method used to better understand the cognitive processes involved in a given activity is the "think-aloud" protocol, in which students are asked to verbalize their thought processes as they perform a task. Researchers typically record and transcribe the verbalizations for qualitative analysis. Analysing these verbalizations provides insight into the cognitive processes that occur during an activity, and can capture strategies, intentions, observations, and emotions [87].

Several studies of student debugging have used the think-aloud protocol [2, 16, 28, 29, 55, 70, 72, 78, 94]. For some studies, the researcher simply observed and took a video or audio recording [94], while others asked prescribed questions such as "What are you doing now?", "Can you explain what you just did?" and "What else have you tried in the past?" [2]. A study comparing third-year students and expert programmers using the think-aloud protocol while debugging a C program found that novices tended to use trial and error when they were unsure how to progress, while experts had a more systematic approach [94]. A think-aloud study of debugging physical systems [37] found that it was important for students to build a mental model that integrated different sources of information.

Think-aloud protocols have also been used to investigate debugging practices used by professional programmers, although there are few such studies. For example, Lawrance et al. [49] observed ten professional programmers debugging a real-world open-source program and used the observations to propose a programmer debugging workflow, and Grigoreanu et al. [32]

used the think-aloud protocol to investigate debugging approaches that IT professionals used in PowerShell.

2.7 Teaching Students to Debug

Li et al. [53] adapted a general troubleshooting framework for use in the debugging domain. The authors suggested that students should be explicitly taught the general steps of constructing the problem space, identifying the fault symptoms, diagnosing the fault, and generating and verifying solutions. Whalley et al. [91] combined this general approach with debugging using the scientific method advocated by Zeller [95] to explicitly prompt students to think about debugging using paper-based exercises. In a study of the student work product, they found that students could use test case information to localize simple bugs, but it was unclear how useful students would find the process in more complex situations. Further work showed that despite struggling to debug effectively, most students did not apply the more structured process they were taught and did not always appear to be aware of their own debugging shortcomings. Almost universally, the participants lacked a formal approach for debugging, describing their approaches instead in vague ways. The authors suggest that better understanding of the gaps between student perceptions of debugging and their practices may help educators to teach debugging effectively [92].

Several experimental studies showed that students benefit from formal debugging training, primarily in a classroom setting [2, 17, 26, 45, 85]. In these studies, the training aimed to explicitly teach novices debugging strategies and about the kinds of bugs they may encounter. The training also allowed novices to gain more experience with debugging. The training included spoken tutorials [26, 45], video- and text-based approaches for explaining debugging strategy [85], and supervised debugging exercises [2]. Notably, two studies found that novices benefited when their most common errors were incorporated in the debugging exercises [2, 45]. Chmiel and Loui [17] implemented the widest variety of teaching activities in addition to debugging exercises: collaborative assignments, reflective memos, debugging logs, and development logs. All of these debugging training studies revealed improvements in student learning, such as their decision-making [85], knowledge [85], and time spent debugging programs [17].

A small number of tools designed to teach debugging have been reported in the literature. Lee and Wu [50] developed a model, DebugIt, and a system called DebugIt:Loop in which students engaged in supervised debugging exercises involving loops. Carter [14] developed ITS-Debug, an intelligent tutoring system to teach debugging, and demonstrated that students using the system learned debugging skills. Lee [51] developed an educational debugging game called Gidget, which was effective at teaching students introductory programming concepts using a bespoke text-based language. Ladebug [60] adapted the PythonTutor system [33] to provide exercises in which students determined which line contained a bug, and were able to edit lines containing bugs to correct them. Miljanovic and Bradbury [67] developed the RoboBUG game to teach students debugging. They found that the game environment was most effective at teaching debugging for students with little previous experience, but some students found the environment frustrating. More recently, Venigalla and Chimalakonda [89] developed G4D, a debugging game in which students debug C code to control a character searching for treasure. Although students were positive about the tool, the impact on debugging was not measured. Overall, there are few tools designed specifically to teach debugging and, although evaluations are generally positive, the evaluations focus on general satisfaction and broad programming skills rather than specific debugging strategies.

There are some observations about student debugging to guide progress in the area. Ahmadzadeh et al. [1] found that good programmers who were weak debuggers lacked an understanding of the program structure. The participants attributed their poor performance to the fact that they were debugging code written by others rather than their own. Similarly, Katz and Anderson [41] found that students have difficulty locating bugs in programs, particularly when debugging code written by others. They reported that students take longer to understand the system when it has not been written by them. However, as difficult as it may be for them, students believe that debugging exercises that incorporate common novice bugs can help their understanding of programming concepts [2].

2.8 Self-regulation and Debugging

While self-regulated learning of novice programmers is not the focus of our investigation, we acknowledge the growing body of work in the area here and briefly reflect on our observations with respect to self-regulated learning later in the discussion. It is now generally accepted that self-regulation plays an important role in learning to program [7, 57, 59, 73] and researchers have reported on positive correlations between self-regulation of learning and programming performance [81]. One study of novice programmers noted that not all types of self-regulation help students prevent errors in their code [57]. The authors reported that the ability to plan and simulate algorithms led to better problem solving and less errors in code while verbalisation and self-explanations led to a small increase in errors. They also noted that novice programmers can self-regulate but do so inconsistently. Most papers, like these, tend to focus on programming and problem solving in general rather than specifically on debugging. However, there are a few papers that focus more on debugging. One such study analysed student reflections [92] on debugging immediately after performing debugging tasks. The students had been taught a process for debugging prior to undertaking the debugging tasks. Like Wert et al. [90], the authors reported that many students had strong emotional reactions to debugging, especially when they encountered difficulties. Wert et al. [90] went on to highlight persistence when programming as a key factor contributing to success for novices. Students react to challenges and associated negative emotions differently, falling into one of two categories: either they persist (are "movers" [72]) or they give up (are "stoppers" [72]).

Whalley et al. [92] also noted that despite students acknowledging that their debugging strategies were suboptimal, their opinions on the utility of a formal structured debugging process, like the one taught, were mixed. This finding is supported by an earlier study in which a simple brainstorming defect localisation strategy was taught [47]. The authors reported a link between using the strategy and number of features implemented. They argued by inference that perhaps only learners with strong self-regulation skills find such strategies helpful and that learners with weaker self-regulation cannot see that their own strategies may not be the best and that they need to use better strategies.

3 METHOD

In this section, we present the context of the study and the method used to collect and analyse the data.

3.1 Course Activities

Participants were recruited from a first-year Python programming course offered at a mid-sized, urban, research-focused university in New Zealand. None of the members of the research team were instructors for the course. Before enrolling in the course, students are expected to be able to read and comprehend code, perform simple code tracing exercises and write solutions to simple programming problems. Students may enter this course directly from high school with an adequate background or may enter after completing a tertiary level introductory programming course (typically also in Python). The course covers a recap of programming principles before moving to data structures in the second half of the course. Due to the varied background of students, it cannot be assumed that all students have been taught debugging prior to enrollment.

Table 1. Participant Demographics

Participant	Sex	Major	Age	Languages
320	female	CS and IS	21	Python
420	male	Geophysics	21	MATLAB, C, C++, Python, R
520	male	CS	18	Python
620	male	CS	19	Python
820	male	CS	19	Java, MATLAB, C, Python
Hugin	**male**	**CS**	**28**	**Python, C++**
Djengo	**male**	**CS**	**18**	**MATLAB, Python**
1220	male	CS and IT management	19	Python
1320	male	Engineering and CS	17	Python, C++, Javascript
JoJo	**male**	**CS and commerce**	**21**	**MATLAB, Python**
2620	male	CS and IT management	18	Python, Javascript
2820	male	Pure maths	22	Python, Java, Javascript, MATLAB, C
3620	male	CS	20	Python, Java

The course was conducted in the summer semester of 2020, which consists of six weeks of formal in-person classes. There were 103 students enrolled in the class during the semester. The course required participation in 20 laboratories conducted as formal two-hour classes. The first 10 minutes provided a presentation that set the context for the activities that followed. The fifth laboratory of the course focused on debugging strategies. It consisted of a presentation that taught a debugging process using a worked example, followed by three activities in which students collaborated on paper-based problems for approximately 20 minutes. The remainder of the laboratory involved students working individually on computer-based debugging exercises (i.e., correcting small programs containing bugs).

3.2 Participant Recruitment Protocol

At the start of the debugging laboratory, the research team extended an invitation to students to participate in this research study. Students were provided with an information sheet describing the research and the processes involved in a think-aloud study. The research team members were available during the class session to answer questions. Towards the end of the lab, interested students were asked to provide their name and email to a member of the research team. The research team then followed up with these potential participants by email and scheduled a think-loud interview session for those who were interested in participating.

3.3 Think-Aloud Protocol

Sessions were conducted in person with one participant and two members of the research team present. One researcher interacted with the participant during the think-aloud while the other took observational notes. Each session lasted approximately 60 minutes and involved (1) a background questionnaire, (2) a familiarisation exercise, (3) debugging exercises, and (4) a brief post think-aloud retrospective interview.

On arrival, the participants were briefed about the think-aloud protocol. We instructed participants to say everything that went through their mind as they read and solved the debugging problems. At this point, participants were given a chance to ask questions. Then, written informed consent was obtained and demographic data gathered related to gender identity, age, prior programming knowledge, and confidence with programming and debugging. This demographic data information is summarised in Table 1. The participants selected for the case study are in bold.

Table 2. Participant Reported Confidence

Participant	Write Code	Trace Code	Debug Programs	Problem Solve	Python Syntax
320	slightly	slightly	slightly	somewhat	somewhat
420	very	very	somewhat	very	very
520	somewhat	slightly	slightly	very	very
620	somewhat	somewhat	somewhat	somewhat	slightly
820	very	somewhat	somewhat	very	very
Hugin	**slightly**	**somewhat**	**somewhat**	**somewhat**	**slightly**
Djengo	**somewhat**	**somewhat**	**somewhat**	**somewhat**	**slightly**
1220	somewhat	very	slightly	somewhat	very
1320	somewhat	somewhat	somewhat	very	slightly
JoJo	**very**	**somewhat**	**somewhat**	**somewhat**	**somewhat**
2620	somewhat	somewhat	somewhat	somewhat	somewhat
2820	somewhat	slightly	slightly	very	not at all
3620	very	slightly	very	slightly	slightly

The participants were asked to rate their confidence in their programming and problem-solving abilities using a four-point Likert scale of very, somewhat, slightly, and not at all. They also answered five questions about their confidence with writing code, tracing code, debugging programs, problem solving, and their knowledge of Python syntax. This information is summarised in Table 2. The participants selected for the case study are in bold.

After completing the demographic questions, screen recording was turned on. The full computer screen and the participants' verbalisations were captured. This recording captured everything the participant and interviewer said, everything the participant typed, all the windows that a participant had open and any interactions with those windows. If at any time participants remained silent for more than one minute, they were reminded to speak out loud about what they thinking or doing. We had no expectation that the participant utterances would all be coherent. In instances in which utterances were incoherent, the observer researcher made a note. These instances were explored in the retrospective interview held at the end of the think-aloud session. In this retrospective interview, the relevant code, exercise descriptions and sections of recordings were visited to support the participants' recollection.

To help participants practice thinking aloud, we provided participants with a simple warm-up debugging exercise that was a variant of a problem encountered early in the debugging laboratory. After the warm-up exercise, the students were given a dictionary processing problem and then three variants of Soloway's Rainfall problem [84]. These activities are described in more detail in Section 3.3.1. After completing an exercise, participants received the next exercise to work on until the interview time was up, the participant decided to stop, or all four problems were successfully debugged. The number of debugging activities presented were dependent on the students, with more successful participants completing all four of the exercises and the least successful participants working on only two of the exercises.

3.3.1 Debugging Activities. Students had some familiarity with the problem context of the dictionary processing and rainfall problems because they were based on the exercises completed during their debugging laboratory. Although students had seen similar code for the dictionary problem, the bugs were new. In the laboratory, students were given the function headers and main method of the rainfall problem but no implementation. Thus, they were unfamiliar with the buggy code presented here.

The warm-up activity is a function that takes as parameters a string representing a piece of text and two integers start and end. The purpose of the function is to create a substring starting at index start and going up to and including index end. The buggy function provided to participants is given here.

```
    def substring(text, start, end):
    substring = ""
    index = start
    while index < end:
        substring += text[index]
        index += 1
    return substring
```

This implementation of the function has a mistake in the while loop, stopping the loop short one index, a mistake marked here in blue. Off-by-one errors have been identified in previous work as being problematic for students [27, 31]. The participants were also presented with test cases and output, which can be seen in the appendix.

The first of the study exercises was a variation of the frequency dictionary problem seen in the debugging lab for the programming course. The function in the think-aloud protocol takes a dictionary with key-value pairs representing people and their votes and creates a new dictionary that represents the number of times each vote appeared in the parameter dictionary.

```
def record_votes(vote_dict):
    votes = {}
    for k in vote_dict:
        if vote_dict[k] in vote_dict:
            votes[vote_dict[k]] += 1
        else:
            votes[vote_dict[k]] = 1
    return votes
```

In this version of the function, there are two errors. The capitalization of the values in the original dictionary should be ignored when recording votes. There is also an error in the if statement inside the loop. The function should be checking if vote_dict[k] is in the new dictionary votes rather than in the existing dictionary vote_dict. Previous work has shown that while errors associated with accessing dictionaries are less common than other errors, they persist longer in student solutions [83]. All of the problematic parts of the function are marked here in blue.

The implementation of the first rainfall problem, called Rainfall A in the remainder of the article, comprised 13 functions. Here, we provide the code for the functions critical to understanding the bugs found in this solution. Note that the doc strings have been truncated from the original and that all function without bugs are omitted due to space constraints.

```
def clean_entire_data(raw_data):
    """Given a list of strings, return a list with valid lists of numbers"""
    clean_data = []
    for raw_month in raw_data:
        month_measurements = convert_monthly_data(raw_month)
        clean_month = clean_monthly_data(month_measurements)
        clean_data.append(month_measurements)
    return clean_data

def get_histogram_row(row_label, value):
    """Generates a single row of a histogram given the label and value"""
    fill_char = 'X' #character to fill histogram bar
    bar = fill_char * int(round(value))
    return f'{row_label}: {bar} ({value:.1f})'

def main(filename = 'raw-data.txt'): #default value
    """ Analyse the rainfall data stored in 'raw-data.txt' """

    #Read the data file
    data = read_data(filename)

    #clean the data by keeping only valid (non-negative) values
    data = clean_entire_data(data)
    #calculate the average values for each month
    averages = get_all_monthly_averages(data)

    #calculate summary yearly statistics
    maximum, minimum, median, mean = get_statistics(averages)

    #output the results
    print_results(averages, maximum, minimum, median, mean)

#Execute the main function of this module
main('raw-data.txt')
```

In this implementation, there is a single incorrect line in the clean_entire_data function. Rather than appending clean_month, which contains the cleaned data, the function appends month_measurements, which retains negative values, marked in blue in the provided code. This produces negative values in the output of the program, which can be seen in the appendix.

In the second version of the rainfall problem, called Rainfall B, the clean_entire_data function is corrected to append the list cleaned of negative values. Instead, there is a mistake in the get_histogram_row function. Rather than rounding a value before converting to an int, the function directly converts to an int which causes a truncation problem. This bug is marked in blue in the code listing. Problems with integer division, which is a similar problem to this bug, have been observed in previous work [5, 27, 36] The two function that differed in Rainfall B from Rainfall A are given next. Again, the doc strings have been truncated and all function that are not directly involved with the debugging activity are omitted.

```
...
def clean_entire_data(raw_data):
    """Given a list of strings, return a list with valid lists of numbers"""
    clean_data = []
    for raw_month in raw_data:
        month_measurements = convert_monthly_data(raw_month)
        clean_month = clean_monthly_data(month_measurements)
        clean_data.append(clean_month)
    return clean_data
...
def get_histogram_row(row_label, value):
    """Generates a single row of a histogram given the label and value"""
    fill_char = 'X' #character to fill histogram bar
    bar = fill_char * int(value)
    return f'{row_label}: {bar} ({value:.1f})'
```

The resulting output, which can be seen in the appendix, produces an incorrect number of values in the histograms printed by the program.

For the third rainfall implementation, called Rainfall C, several functions were completely rewritten and five separate bugs were introduced. Again, only the crucial function have been provided here, and the doc strings have been shortened.

```
def clean_monthly_data(data):
    """return a list of all the non-negative values in data"""
    i = 0
    while i < len(data):
        if data[i] < 0:
            data.pop(i)
        i += 1

def get_all_monthly_averages(data):
    """Calculate the monthly average rainfall and return a list of averages"""
    monthly_rainfall = []
    for month in data:
        average = get_single_month_average(month)
        monthly_rainfall.append(average)
    while len(monthly_rainfall) <= 12: #in case the file doesn't have all months
        monthly_rainfall.append(0)
    return monthly_rainfall
```

```
def get_median(averages):
    """Given a list of numbers, return the median value of that list"""
    averages.sort()
    size = len(averages)
    if size % 2 == 1:
        median = averages[size // 2]
    else:
        median = (averages[size // 2 - 1] + averages[size // 2]) / 2
    return median

def get_statistics(averages):
    """Return the maximum, minimum, median and mean rainfalls"""
    maximum = max(averages)
    minimum = min(averages)
    mean = get_mean(averages)
    median = get_mean(averages)
    return maximum, minimum, mean, median

def main(filename = 'raw-data.txt'): #default value
    """ Analyse the rainfall data stored in 'raw-data.txt' """

    #Read the data file
    data = read_data(filename)

    #clean the data by keeping only valid (non-negative) values
    data = clean_entire_data(data)

    #calculate the average values for each month
    averages = get_all_monthly_averages(data)

    #calculate summary yearly statistics
    maximum, minimum, median, mean = get_statistics(averages)

    #output the results
    print_results(averages, maximum, minimum, median, mean)

#Execute the main function of this module
main('raw-data.txt')
```

The bugs in this implementation are:

- In the clean_monthly_data function the list data are not returned. Further, the variable i should only be incremented if the pop does not occur.
- In the get_all_monthly_averages function, the while loop that appends extra zeros should be deleted.
- In the get_statistics function, the variable median is being assigned to a call to get_mean when the function that should be called is get_median.
- In the get_statistics function, the variables are returned in the wrong order. The return statement should return maximum, minimum, median, mean.
- The get_median function is sorting a list, which impacts the printing of the histogram. A copy of the list should be sorted to avoid this.

Problems with return values from functions have been observed and analysed frequently in previous work [4, 5, 42, 88], although the shuffling of types that occurs in the get_statistics function is only possible in a language such as Python. Providing an incorrect number of parameters [63] or providing correct parameters but in the wrong order [13] has also been observed in previous work, both of which are related errors. Finally, erroneously modifying a list that has been passed as a parameter is a bug related to pass-by-value versus pass-by-reference variables, something that has also been studied previously [42, 61].

All of the bugs have been marked in blue in the code listing. Note that an additional bug is that clean_monthly_data is missing a return.

Table 3. Progress by Participants on Debugging Activities

Participant	Warm up	Dictionary	Rainfall A	Rainfall B	Rainfall C
320	Completed	Stopped	Completed	Completed	Stopped
420	Completed	Stopped	Completed	Completed	Stopped
520	Completed	Gave up	Completed	Out of time	Not started
620	Completed	Stopped	Completed	Not started	Not started
820	Completed	Stopped	Completed	Completed	Gave up
Hugin	**Completed**	**Gave up**	**Gave up**	**Not started**	**Not started**
Djengo	**Completed**	**Completed**	**Completed**	**Completed**	**Completed**
1220	Completed	Completed	Completed	Stopped	Not started
1320	Completed	Completed	Completed	Completed	Stopped
JoJo	**Completed**	**Completed**	**Completed**	**Completed**	**Not started**
2620	Completed	Completed	Completed	Completed	Not started
2820	Completed	Completed	Completed	Completed	Stopped
3620	Completed	Completed	Completed	Completed	Stopped

3.3.2 Participant Progress and Study Selection. The progress of each participant in the study on the activities is presented in Table 3. The participants marked as having completed an activity were able to correct all of the bugs in the provided code. Participants who were stopped by the research team were making insufficient progress on the debugging activity and did not show signs of being able to make further progress. Participants marked as giving up indicated that they wished to stop working on the specified activity. Participants who are marked as running out of time reached the end of the time allocated for the think-aloud without completing the activity but were still appearing to make some progress. Those marked as not starting ran out of time before the activity could be started. The participants selected for analysis in the study are in bold.

All participants completed the warm-up exercise. Only one participant (Djengo) was able to complete all of the debugging activities. Five of the 13 participants were able to complete all but one of the activities (1320, JoJo, 2620, 2820, and 3620). Four of the participants were able to complete two of the four activities (320, 420, 820, and 1220). Two of the participants were able to complete one of the four activities (520 and 620). One participant, Hugin, was unable to complete any of the study activities.

When selecting participants for analysis in this case study, the goal was to choose participants who covered the spectrum of ability across the participant pool. The participant who completed all of the activities (Djengo) was chosen. One participant was chosen from the group who completed all but one activity (JoJo). The participant who was not able to complete any study activities (Hugin) was also chosen. In the latter case, Hugin was chosen because his spoken English was stronger than either participant who was able to complete one study activity. We felt that choosing a participant with a poorer performance but better English skills would provide more insight for a think-aloud study.

3.4 Reflexive Thematic Analysis Process

The purpose of thematic analysis is to develop a shared understanding of the underlying patterns of meaning across a dataset. These patterns are generated through a rigorous process of data familiarisation plus an iterative process composed of data coding and theme development. Here, we use a reflexive thematic analysis [11, 12] in which theme development is directed and both reflects the content of the data and examines the deeper underlying assumptions within a theme. In this form of thematic analysis, the themes are the end point or outputs of the analysis. Both the analysis techniques and philosophy of research are qualitative, making this Big Q [44] thematic analysis. This approach means that positivist approaches, such as the creation and use of a fixed code book to code the data and measurement of coding agreement, are not part of our analysis. Choosing this approach means that there is a focus on researcher immersion and deep engagement with the data, codes, and themes in a flexible, open exploratory process. Coding is fluid and codes evolve throughout the analysis process as the researcher's conceptions of the data grow though multiple passes of the data. The aim is to represent the researcher's conceptualisation of the data. The depth of interpretive engagement means that the analysis and themes reflect the researcher's cultural membership, theoretical assumptions, and epistemological viewpoint. The following points outline the six phases in our implementation of Braun and Clarke's [11] reflexive thematic analysis approach.

(1) **Familiarisation with the data** – Each researcher independently recorded a timeline of significant events by viewing and reviewing the video and reading and rereading the interview transcripts. This phase allowed the researchers to become immersed in the data. The complete think-aloud data for all three participants was transcribed and analysed.
(2) **Coding** – Each researcher independently coded significant events or anchors within the data. In this phase, a complete set of labels was derived that were then collated by consensus. An in-depth discussion between the researchers informed an iterative revision of labels that involved the splitting, renaming, and merging of labels from the individually derived codes to build a single set of labels. As this process was undertaken, the data were repeatedly revisited and reinterpreted as needed to ensure that the labels were representative of the researchers' understanding.
(3) **Generating initial themes** – The researchers collaboratively generated candidate themes through an iterative refinement process. In practice, this initial theme-forming phase overlapped to some extent with the prior phase in which refinement of the labels led to better understanding of the data and its associated underlying assumptions. Themes then began to be conceptualised.
(4) **Reviewing themes** – This phase involved merging, splitting, and discarding themes as the viability of each theme was evaluated by the researchers against the research aims and questions. During this phase, approaches to grouping the themes into overarching thematic concept areas was explored and consensus reached on a final classification schema.
(5) **Defining and naming themes** – This phase involved finalising the name, scope, and focus of each theme through consensus.
(6) **Writing up** – This final phase tells the story of each theme through analytic narrative and extracts from the interviews. Finally, this narrative was analysed in the context of existing literature in the area.

It is important to note that while these phases are presented sequentially, each phase builds on the prior phases. This means that the analysis is a recursive process, with movement back and forth between the phases.

Several challenges arose during the analysis. Some participants did not articulate their thoughts readily and needed to be prompted — in such situations, we were occasionally unable to determine their motivations for a given action. Additionally, we needed to be careful not to infer a particular cause for a behaviour we observed. For example, we often wanted to ascribe some form of hypothesis generation to an utterance or an observed action, but there was not sufficient evidence for this. This meant that when we attempted to reach consensus on an observation labelled as related to hypothesis generation, we ended up deciding on a lower-level tactic such as bug localisation. In the end, the hypothesis label was removed from the analysis.

Table 4. Summary of Themes and Sub-themes

Comprehension (see 4.1)	Understanding tools (see 4.1.1) Understanding the requirements (see 4.1.2) Understanding the issue (see 4.1.3) Understanding faulty code (see 4.1.4)
Activities (see 4.2)	Speculating cause (see 4.2.1) Bug localization (see 4.2.2) Code modification (see 4.2.3)
Workflow (see 4.3)	Context switching (see 4.3.1) Systematic (or not) (see 4.3.2) Sustained (or not) (see 4.3.3)

In Section 4, we present the themes and their narratives. The analysis of narrative and how it relates to the existing literature is presented in Section 5.

3.5 Acknowledging Researchers' Positionality

Coding in reflexive thematic analysis is an active and reflexive process that is inevitably seen through the lens of the researchers. This research approach involves a reduction of participants to their verbal utterances and the researchers' interpretation of those utterances. Thus, it is important to acknowledge the researchers' own perspectives and positioning. Two of the researchers were women and one was a man. All three researchers are experienced computer science education researchers and teachers with around 90 years of combined experience. Over the years, all three researchers have taught introductory programming using a variety of programming languages and pedagogies. Two of the three researchers were visiting researchers from other institutions and were "outsiders" [18]. They did not have the same connections to the institution as the other researcher. While the third researcher was from the institution where data was collected, the researcher was not involved in the design or delivery of the course. One of the "outsider" researchers was learning Python for the first time at the time of the study, giving the research team two different lenses of interpretation, one of which was closer to the experiences of the participants. This insight at times during the analysis challenged the unintentional assumptions of the other researchers. The other researchers brought years of experience of teaching and programming in Python. Their experiences provided a broader overview and deeper insight into the difficulties faced by students learning to debug Python.

4 RESULTS

Our thematic analysis resulted in three themes — Comprehension, Activities, and Workflow — and ten sub-themes. These themes and sub-themes are summarized in Table 4.

4.1 Comprehension

This theme collates participant practices related to understanding a debugging problem. The theme includes four sub-themes of observations related to understanding the tools needed for debugging the problems, understanding the requirements of the code for each problem (i.e., intent of the program), understanding the buggy code, and understanding the problems that arise as a result of running the buggy code.

4.1.1 Understanding Tools. This sub-theme encompasses instances in which the participants demonstrate a lack or presence of knowledge and understanding of the tools needed to be able to troubleshoot and debug code. These tools include the programming language and the development environment. This includes skills such as running and executing code in IDLE and approaches to navigating the Python documentation or the use of the Internet to access programming language information and relevant code examples.

Programming language: All three participants showed some fragility with respect to the Python language constructs needed for the tasks in this study. However, they all had basic programming syntax knowledge and were able to print and use familiar functions. The difference between the participants lay in differences in their degree of knowledge of Python and in their ability to cope with any gaps in their knowledge. Even the strongest student, Djengo, struggled a little with dictionaries and was unclear on key versus value saying, *"Uh, so if two keys hit the same value, or is it the two values have the same key, then they will be added to the output?"* It is worth noting here that while dictionaries were covered in the course from which participants were recruited for the study, there was little hands-on practice in programming using dictionaries. Hugin repeatedly remarked that he was *"not good at dictionaries"*. He was unable to access values from a dictionary and had no strategy for finding out how this could be done. He did not attempt to read the documentation or search the Internet for information. After 23 minutes, Hugin gave up on the dictionary problem, stating that he was *"quite stumped"*.

Two of the participants, Djengo and JoJo, failed to investigate library functions in the provided faulty code that they were unsure of. In addition, there were incidences in which reading up on a library function or data structure would have helped the participants to understand the buggy code. For example, Hugin confused keys and values in the dictionary problem and did not understand how the `sum()` function worked in Rainfall A.

Only Djengo was able to successfully cope with fragile Python language knowledge by searching the web for help. When working on the Rainfall B problem, he was unsure of the rounding functions provided in Python, remarking: *"so possibly it would be ceil, I think ... is it ceiling? I feel like that's a Python function, but it is not going purple, which would indicate to me that it's incorrect"*. IDLE on the university machines has syntax highlighting configured so that known functions are rendered in purple text. In this case, the call to `ceil` was not highlighted. Djengo then ran the code and checked the IDLE output and noted *"maybe it's [ceil] not defined, true [reading the* `NameError`*], it is not defined"*. Djengo then shifted to a browser and searched *"round up in python 3"*. After reading two Stack Overflow entries about rounding, he selects a third site and realises *"Oh, it's math.ceil that's the issue. Okay. Just got to import"*. While using `ceil` did not solve the problem, employing `ceil` gave Djengo additional information that quickly led him to a correct solution.

There were a few instances in which the other participants unsuccessfully searched for a library function that they felt should exist and wanted to use. For example, in the dictionary problem, Hugin tried to find a function to capitalise the first letter of a word online, noting *"So I'd [sic] need to find the syntax for 'how to capitalize the first letter of a string in python'"*. He read the first entry on the search results *"So you go string.capitalize() ... first character is in lowercase ... I don't know if that lowercases, everything else"*. He failed to recognize that he really wanted a function to convert the entire word to lowercase.

Environment: We observed several environmental factors that affected a student's ability to understand and debug the program code, such as the spatial layout of information, familiarity with the integrated development environment (IDE), and efficient use of the environment.

At the start of each exercise, Djengo **arranged the windows** so that all of the information he needed to work on the problem was visible. JoJo did not consistently **organize** his **screen real estate**, but when solving Rainfall A, he positioned the windows to allow the Python file and input file to be viewed simultaneously. JoJo also tended to add blank lines to the code, mentioning

"I like to have spaces just to make it easier to read". There were no instances when Hugin worked to optimize the visibility of the various open windows or code layout.

Both Djengo and JoJo were able to **use IDLE to execute and test code** with reasonable competency. They could run code, translate the test cases provided as text files, execute the tests and interpret much of the test output. They could easily cope with syntax errors and recognise the difference between syntax errors and semantic errors as they occurred during debugging. Djengo noted *"Um, usually the error messages are pretty helpful, some IDEs even let you like click on it and it will go directly to the line, which is very useful"*. Interestingly, the participants tended to rely on printing to debug rather than using the debugging tools provide by IDLE. However, Hugin demonstrated limited knowledge of IDLE, and of the process of executing code in an IDE. Importantly, he did not know how to call the function he was debugging using the test case information provided and persisted in making changes without executing any code. A member of the research team needed to point out how to copy the example tests in the problem description into the IDLE window where they could be executed.

We also observed **processes that were more or less efficient**. For example, Hugin always killed the IDLE window rather than reloading the file and re-running the code. In contrast, when checking the histogram length while debugging Rainfall B, Djengo cut and pasted the output from the code to compare with the expected output rather than manually counting the Xs as the other participants did. Djengo used this efficient approach strategically and only when the number of Xs was too long to readily compare the histogram bars in situ (see Appendix A.3 for histogram representation), commenting *"there are different amounts of Xs. I can see here [highlighting with the cursor actual output for March] this has four and that [expected for March] has five ... some of them have more than they should ah, um, but its not consistent ... I'm going to copy this long one and paste it in here for a second ..."*

4.1.2 Understanding the Requirements. This sub-theme encompasses observations that relate to understanding the intended requirements of the program.

Reading the provided specification: Both Djengo and JoJo read function docstrings before examining a function's code. All three participants read the programs' specifications but with different attention and comprehension observed. Djengo, who was the best at debugging, tended to begin to tackle a problem by first reading the entire program specification and spent time reading test cases and input data. He tended to favour reading over printing to understand code and did not use printing to support his reading. For example, when first encountering Rainfall A, he first looked at the test case's expected output and mentioned *"So this looks like some sort of histogram would be my guess"* but then quickly shifted to reading the specification, periodically cross-checking with the test case and information about the input file data supplied in the program header comment, stating *"... it'll be useful to know how the data is stored so that how it's going to be read. Oh each line is a month, I think... with the days separated by spaces ... uh, some lines are left blank. If no data was collected that month. ... Hmm, the data collected is intermittent. So the number of measurements in different months may be different"*. In contrast, JoJo's reading tended to be inconsistent, quick, and cursory. He appeared to prefer to jump into debugging and frequently returned to the specification to check his understanding of the intent of the code. Hugin also read the descriptions given for functions but he did not demonstrate understanding of the function's purpose.

Using test cases to understand intent: All three participants used the test cases as examples to understand the purpose of the program. JoJo started by focusing on the test case function call and its output: *"Okay. So, the first thing, I guess I'd looked at the, um, thing that's being run ... So it says d1 and then there's a bunch of names ..."*. JoJo went on to express a loose theory about the code purpose:

"Oh, Okay, it's kind of a count, so it counts the number of times the name is in the dictionary." He then read the specification and returned to the test case highlighting the dictionary's first key/item pair and uttered *"Oh, so it's someone choosing someone else?"*

4.1.3 Understanding the Issue. This sub-theme comprises activities that relate to building an understanding of the issue(s) that arise when the buggy code is executed using the test case data and how issue(s) identified relate to the intent of the code.

Reading error messages: Of the three participants, only Djengo was observed reading syntax error messages and using them to determine problems in the code. In the warm-up exercise, he attempted to run the test case without supplying the data to the function *"So, first I guess I will run it to see if there are any errors"*. He then copied `substring(data,0,4)` from the test case file into the IDLE editor and ran the code and got a `NameError`, uttering *"Uh, so it says name data is not defined. So, I would look for, uh, what line is that? Sub-string data? Zero four. All right. 'Cause I haven't got my data. That's probably not the intended bug."*

Identifying symptoms: We observed all three participants comparing the actual and expected outputs of the test cases provided to identify bug symptoms. All three participants were able to identify the symptoms. For example, Hugin concluded from examining the test case outputs for Rainfall A that *"So it looks to me by comparing them they are wrong they are off. This one's negative."* Djengo, after reading the code specification, always looked at the test cases. In the dictionary problem, he immediately noticed that the actual and expected outputs were the same and so moved on to the second test case. *"So it's expecting Simone 2 and Andre 1, but it's getting Simone and Simone. So the issue here is that, um, the two different values, Simone and capital Simone [SIMONE] are being treated as different when they should be treated as the same thing"*. Here, the symptoms observed are the incorrect vote count and the incorrect capitalization of the "Simone" key in the actual output. After examining the actual and expected output and the doc strings at the top of the Rainfall A problem, JoJo concluded that the issue with the code was the negative values, saying *"so, since I'm getting negative values so that should be a problem here."*

Identifying the issue from test case data: Both JoJo and Djengo were able to generalize based on the provided test case data. For example, JoJo quickly realised the dictionary problem was related to the case of the letters in each of the words in the input data. He noted the differences between the second test case's actual and expected outputs: *"Okay. Oh yeah. And it's considering the two Simones as different because one is capitalized [Simone] and one is lowercase [simone] but it should be the same. Okay, I see."* Hugin examined the first test case, correctly noting that there was no difference between actual and expected output. When examining the second test case, he noted *"So, it seems to be that it's registering keys as different keys if they're capital or if they had lowercase."* Unlike the other two participants, while Hugin attempted to generalize from a single case to a more general problem, his generalizations were poorly focused. For example, he summarised that *"the keys are different if they have a different case"*. This is true but doesn't really get to the crux of the problem.

When tackling the Rainfall A problem, Djengo was able to generalize based on the monthly rainfall output being too low that the Rainfall A program was not ignoring negative values. This comparison between actual and expected output then quickly led Djengo directly to speculations about the cause of the bug: *"So, what happens when it runs? Um, the actual and expected output differ quite a bit. Um, so, what are the issues? So, the first issue seems to be that the histograms have the incorrect amount of Xs. Um, I assume because it's getting the incorrect data. So, the monthly high is correct, the monthly low is incorrect because it's not ignoring negative values"*.

4.1.4 *Understanding Faulty Code.*

This sub-theme groups activities related to understanding how the buggy code has been built, including understanding the program flow and building a mental model of the buggy code.

Understanding program flow: Attempts to understand the flow of the code were made. Both Djengo and JoJo made attempts to follow the execution path by printing function inputs and outputs. When Djengo was trying to understand the flow of the program, he first read the main function and then printed the output of each function in order to determine the flow of data and location of the error. At one point while working on Rainfall A Djengo comments *"So, um, if I didn't know what data was at a particular instance before it entered or exited a definition, I'd print data to check what it is."* This suggests that his approach to understanding the data flow changes with the complexity of the problem.

In contrast, Hugin does not attempt to understand how the functions in the Rainfall A problem fit together before commencing debugging. He made no attempt to trace the program flow. He was observed jumping around in the code seemingly at random and reading bits of code in isolation without knowing how the pieces were related. In fact, Hugin never examined the main function of the rainfall problem. When Hugin was working on Rainfall A, he mentioned that `clean_monthly_data` should be doing something, but couldn't see how it would change the data, indicating that he could not see the overall flow of data through the problem. Hugin alone attempted to understand the content of specific variables, commenting *"I've tried [to locate the bug] by printing the averages that I'm getting off [incorrect]".*

Building a mental model: Djengo was seen building a mental model of the program by understanding how the parts connect. He also tended to summarise in his own words the purpose of these parts using the output of functions to connect and verify the purpose of the parts. For example, when looking at the `clean_monthly_data` function in Rainfall A, he summarised the function's purpose: *"So, this is getting all the non-negative values".* Djengo, however, did not always build a full mental model of the program and its parts before attempting to fix the bug. In Rainfall A, he focused on looking for the location of the code that read in the data without scanning/reading through the program to get an overall idea of structure. As a result, it took a while before he decided to read the main function. In contrast, Hugin was often unable to connect the results the code was producing to what he was seeing in the actual and expected test outputs. JoJo seemed unable to connect the function documentation to the function behaviour.

Checking: We observed participants checking or verifying their understanding of a particular aspect of the buggy code. This included checking the documentation or running code to see how a function works. We observed both static and dynamic forms of checking.

Static activities included renaming variables, rewriting code, reading code, reading comments, and reading the provided test case output. JoJo often renamed variables, although the renaming did not always produce improved understanding. For example, when trying to understand the dictionary problem, he changed the name of the variable controlling the loop from `k` to `names_pair`, commenting that it was a better description for the value in the variable. Unfortunately, that variable is actually holding the keys in the parameter dictionary, not key-value pairs. JoJo justified the modification by saying *"It might be waste of time, but I like to do it if I have time just to make it easier for me so I understand what's going on more clearly".* After reading the second test case for the dictionary problem and comparing the actual and expected output, Djengo was able to speculate on a solution: *"Uh, so you'd want to put them all into a sort of unified format so that, regardless of case, they all get put into the same thing."*

Dynamic interactions involve activities such as printing, executing test cases, and creating new test cases that are used by the participants to build an understanding of the code they are troubleshooting. For example, when working on Rainfall A, Djengo was attempting to locate the issue with negative values in the output. He commented: *"All the time I find that it's just easy to print things to see if it's actually getting what you'd expect it to get."* Later, while working on the same problem, he said: *"So, what I might do is just print the data here to see if that is actually where it it occurring."* One example of testing the buggy code with the test case data provided to verify the test case's actual output is when JoJo executed the code using the provided test cases and mentally mapped the input test data (`d2`) to the function parameter (`vote_dict`), uttering *"Um, so, now to check the code, so, it takes* `vote_dict`*, which is the* `dict_2` *in this case."* JoJo used print statements a lot to understand code. He focused on dynamic behaviour during code execution, which allowed him to see variable states, rather than reading documentation and code.

4.2 Activities

This theme includes the debugging activities related to the steps, strategies, and processes for debugging the faulty code that the participants were provided and includes speculating about the cause of a bug, bug localization, and code modification.

4.2.1 Speculating Cause. This sub-theme encompasses activities that indicate an attempt to speculate about the cause of a bug. JoJo and Djengo both speculated about the cause of bugs in the provided code, although the degree with which they were systematic about their speculation differed. Djengo did not debug until he had speculated about what was wrong. Djengo also speculated first and then tested his hypothesis. However, the speculation lacked precision, and he formed only a partial theory about the bug cause. For example, in Rainfall B, Djengo identified a problem in number of Xs in the histograms. But he didn't try to figure out the specific issue (i.e., truncation) and was happy to locate and correct code knowing that it was *"something to do with rounding"*. The closest he came to forming a hypothesis was to note *"and it might have to round down or up. Maybe it's doing that wrong."*

JoJo was less specific about his hypotheses, for example, indicating that the problem in the dictionary function was with capitalization but not being specific about whether the capitalization was in the keys or the values. JoJo's hypotheses were often only partially specified or, in some cases, not specified at all. For example, when debugging Rainfall A, he was unable to locate the problem with the actual output and decided to examine the code. He then stopped himself, remarking *" ... since I can't tell the difference between thing I was like, maybe I could check the code, but then if I don't know what's wrong, I don't know what to look for in the code. ... Let me go check the differences again."* After examining the actual output again he commented *"it's one short so it's a rounding error ... So it's printing one less x for the rounding"*. He did not name a specific issue regarding the rounding and its relationship to the histogram before moving back to examining the code.

When Hugin developed hypotheses about bugs, they were incomplete and/or flawed. For example, for one of the rainfall problems, he speculated *"... it looks like it isn't reading the negative values properly"* but he was not precise about which function might be causing the problem. Even when his speculation was correct, for example, in the warm-up problem where he suggested that <= should be used rather than <, he failed to follow through on testing the hypothesis by running the corrected code.

4.2.2 Bug Localization. Participants used observations about the actual versus expected output and understanding of the purpose of functions to help them locate potential bugs in the code. Their efficiency and effectiveness at doing this varied, however.

When working on the Rainfall A problem, Djengo attempted to locate where the problem with negative numbers was occurring. He decided that adding print statements would help, saying he was *"... just zeroing in on where the issue is"*. Unfortunately, he stopped just before the line with the bug in the `clean_entire_data` function and moved to examining the rest of the code. After scrolling briefly, he commented *"I feel like I'm looking in the wrong place"*. He moved to the main function and commented that he was *"just working my way backwards through the codes to see if I can find where the error might initially occur"*. After a few minutes, this backward approach brought him to reconsider the `clean_entire_data` function, where he discovered the bug. Upon discovering the bug, he remarked: *"That took a lot longer than I expected"*. In the Rainfall B problem, Djengo observed an error in the actual output, noting *"so, it looks like everything is correct apart from the amount of Xs being printed"*. He immediately concluded that he should consider the function that printed the histogram, using his knowledge of the structure of the program.

JoJo was somewhat effective in bug localization, actively searching for functions based on their expected behaviour. When working on Rainfall A, he observed that the actual output had negative values when it shouldn't and then commented: "`clean_monthly_data`. *Given a list of numbers return a list of all non-negative values. Oh, so, since I'm getting negative values, so, that should be a problem here."* In the Rainfall B problem, JoJo concluded that the issue was with rounding and the display of the histogram. He then began reading the documentation and code for the functions involved in the printing of the histogram.

Hugin was the least effective in bug localization. In the dictionary problem, he was able to identify capitalization of keys as a problem and then move to the correct location in the function, remarking *"So, I'd look in here where it is adding the keys, which is, this part here on the else statement."* When debugging the Rainfall A program, Hugin actively sought the location in the code that dealt with negative values. However, he was inefficient in trying to find the buggy code, scrolling through the provided code rather than systematically searching for particular functions. He repeatedly talked about functions passing information to other functions, but he never looked at the main function, where each of the functions were being called and information was shared between the functions. Hugin also did not appear to have any overall strategy for locating bugs in larger programs.

4.2.3 Code Modification. This sub-theme encompasses the approaches and techniques that participants used when modifying the code while debugging. The issues observed when the code was modified fell into three clear types of activities: trial-and-error, rewriting code, and deleting code.

In general, the participants were not systematic about code modification. Djengo would begin to trace code to understand bugs but would change the code before completely reading it. He would also generate incomplete hypotheses about bugs, such as that the number of Xs in the histogram for Rainfall B were incorrect, but did not try to figure out that the specific issue was truncation before moving on to correct the code. In modifying code during debugging, JoJo would introduce new bugs, for example, a test using greater than rather than greater than or equal to or the use of the equality operator rather than the assignment operator. He would at times then fail to find his own bugs in the modified code. Hugin had similar problems as JoJo, except that the new bugs were more significant, for example, looping through an empty dictionary or dividing by a nonexistent variable.

Trial-and-error. All of the participants seemed to take a trial-and-error approach to code modification to some degree. Djengo would change unfamiliar code without knowing what it does or what the change might do, for example, changing += using a list to append, which is equivalent. He remarked on occasion that he was making the changes "just in case". However, both of the weaker debuggers took a trial-and-error approach more often. JoJo was particularly likely to make speculative changes to code. He would make edits to code without having a clear idea why those edits were being made. JoJo also made changes to functions in the rainfall problem that replicated existing functions, showing that he made changes before understanding the structure of the entire program. JoJo also commonly removed edits from code without understanding why the edits were failing to correct the problem. However, Hugin was the most egregious in making unsubstantiated edits. For example, in the dictionary problem, Hugin changed a key to lowercase but changed the key in the original dictionary rather than the key in the dictionary being built by the function. Because Hugin had weak testing ability, he was unable to detect this issue. In Rainfall A, Hugin observed that the actual output did not match the expected output. Thus, he changed an operator from greater than to greater than or equal to just in case that made a difference. The error, however, had nothing to do with the operator. Hugin also failed to follow through on failed bug fixes, reverting to the original code rather than exploring why the fixes did not work.

Rewriting code. JoJo's first approach when encountering code that he did not understand was to rewrite it. *"And I'm not 100% sure of how this is working. So, I'll just do what I need to do and I'll just comment and rewrite it the way that I'd do it"*. He would delete provided code and rewrite it when he could not see any error with the code but believed that there might be a problem. Only when his rewrite of the code did not produce any change in the behaviour of the function would he move on to consider other possibilities for bugs. Interestingly, he drew a connection between having time to rewrite code and his confidence in debugging, commenting *"that's one thing I find with coding is I'm more confident if I have lots of time, but in a test condition where you don't have enough time to rewrite the whole code or you're not allowed to then it's like, yeah ..."*

In the Rainfall A problem, Hugin opted to rewrite code rather than spend time attempting to understand the existing code. When trying to identify why negative values were appearing in the output, he focused on the `get_single_month_average` function. Believing that the problem had to do with the built-in `sum` function used in that function, he changed the function to use a for loop to add up the non-negative values in the list provided to the function. Since his modification was equivalent to the code he deleted (and because the error was not in that function), the change did not have any impact on the results, which he discovered by running the program.

Deleting code. When working on the Rainfall A problem, JoJo remarked on his inefficiency with respect to print statements, noting *"maybe my method isn't that great because I'm wasting a lot of time going back and deleting print messages that I put earlier, and then when the code is very long like this, I forget where it was ..."*.

As mentioned in the trial-and-error section, Hugin often removed changes to code when they did not appear to have helped. Unfortunately, because his testing skills were weak, there was often no evidence indicating that the changes were or were not effective. For example, while trying to debug the dictionary problem, he changed the for loop over the parameter dictionary to move over the items (e.g., the key-value pairs) rather than the keys but deleted the edit before either using it or testing any possible change as a result of making the edit.

4.3 Workflow

This theme focuses on observations about debugging behaviours and how systematic the students were across the debugging tasks, including the sub-themes of context switching, being systematic (or not), and working in a sustained way (or not).

4.3.1 *Context Switching.*

This sub-theme resulted from participants switching between multiple sources of information and often involved comparing or referencing between these sources.

The comparing or referencing was sometimes systematic and other times done in an ad-hoc manner.

When Djengo began work on Rainfall A, he first observed that the output showed a histogram. He moved from looking at the output to reading a doc string in the file, observed that the problem involved reading from a file, and noted *"it'll be useful to know how the data is stored so that I'll know how it's going to be read"*. He examined the structure of the input file and commented *"so, each line is a month, I think... with the days separated by spaces"*. He then read in the doc string that lines may be blank if no data was collected that month, moved the cursor to select the text file with the data, and verified that there were blank lines in the file. He then moved to considering how the actual and expected output differed, focusing on the monthly high and low values. He observed that negative values were not being ignored.

JoJo was observed to be shifting frequently between reading, printing, and checking outputs when trying to build understanding in more complex problems. Unfortunately, his context switching was not always systematic. Like Djengo, JoJo began the Rainfall A problem by examining the sample output provided. He saw that in the actual output there was a negative value whereas in the expected output there was not. He switched to reading the doc string at the top of the file containing the code. Once he concluded that the problem was with negative values, he started reading the first function in the file, which was `clean_monthly_data` and, after reading the doc string, concluded that the problem must have been in that function. However, before he could verify anything about the function, he decided that he needed to learn more about how the file was read and jumped to a new part of the program, saying *"so, the reason why I'm looking for opening the files ... is that ... you need to open the file first to get the file contents out before you can clean it ..."* JoJo switched from the function that was cleaning the data to one reading the data without having a good reason to do so.

4.3.2 Systematic (or not). How systematic the participants were varied widely. Some of the less systematic behaviours seen in participants include jumping to conclusions, trial and error, and guessing.

Djengo was the most systematic in his debugging approach and became more systematic with increasing problem size (lines of code and number of bugs). In a few instances, Djengo was hasty and "fixed" code without due consideration. He then had to go back and reconsider the changes he had made to the code. For example, in the Rainfall C problem, he added a += operator in a location that would not work. When that approach failed, he tried instead to use append on the list. However, he did test these changes and other changes during debugging and was able to go back and rectify the issues he had introduced to the code during his early debugging attempts. While Djengo exhibited some good techniques, he didn't put these together into a consistent systematic approach. For example, he did not systematically print and check input and output from functions. Djengo tended to trace the code to understand the program flow and functions but often changed the code before he had completed reading and tracing it. At times, he failed to follow through on an idea. At times, Djengo was observed jumping from function to function in Rainfall A and not reading the code or documentation before switching to another function.

JoJo articulated the need to identify the problem before debugging but lacked a methodological approach to locating bugs. He sometimes looked for the bug source before reading the program or the requirements. He would start reading but then abandon that process part way through. At times while working on the rainfall problem, JoJo did not understand the structure of the entire program but made changes to parts of the program anyway. In one instance, JoJo made changes that replicated functions already present elsewhere in the program. Overall, JoJo lacked a methodical approach to locating bugs. In Rainfall A, he was observed jumping from function to function

Table 5. Effective Practices

Practice	**Examples**
Theme: Comprehension	
Using print to examine variable contents	Djengo (4.1.4)
Seeking help in online documentation	Djengo (4.1.1)
Using environment cues, such as syntax colouring	Djengo (4.1.1)
Organizing work space to maximize visible information	Djengo and JoJo (4.1.1)
Improving readability of code with superficial changes	JoJo (4.1.1, 4.1.4)
Executing tests to determine whether code is correct	Djengo and JoJo (4.1.1)
Reading docstrings and requirements before code	Djengo and JoJo (4.1.2)
Identifying differences between expected and actual output	All (4.1.3)
Generalizing from specific cases to the issue	All (4.1.3)
Speculating the cause of an issue	Djengo and JoJo (4.1.3)
Reading error messages	Djengo (4.1.1, 4.1.3)
Tracing code and understanding control flow and data flow	Djengo and JoJo (4.1.4)
Building a mental model of the program	Djengo (4.1.4)
Verifying understanding of program state	Djengo and JoJo (4.1.4)
Theme: Activities	
Speculating on the cause of an issue	Djengo and JoJo (4.2.1)
Using print to examine variable contents	Djengo (4.2.2)
Using system knowledge to localize bugs	Djengo and JoJo (4.2.2)
Theme: Workflow	
Cross-checking to verify understanding	Djengo and JoJo (4.3.1)

checking everything but the last line in a function. JoJo also failed to read the main function to understand how functions work together. He did not apply effective debugging processes consistently.

4.3.3 Sustained (or not). Of the three students, Djengo was the one who most clearly demonstrated sustained activity. He continued working even when unsure of what he was doing. When his working hypothesis about a bug cause or location was shown to be false he stopped checking data flow and this seemed to affect his confidence. However, he continued working and wanted to complete the tasks he had been given, working beyond the time allocated for the debugging session.

Hugin demonstrated the least amount of sustained activity. After working on the dictionary problem for about 10 minutes, he said *"Okay, so ... I'm quite stumped here to be honest."* The researcher offered to allow him to move to the next problem or to review the debugging process taught in the class as a help in moving forward. He opted for the debugging process, spent less than 2 minutes reading the process, and tried to use it before commenting *"Do you mind if we move on to a different one?"* He similarly stopped working on the Rainfall A problem.

4.4 Summary

Table 5 summarizes the effective debugging practices observed. Table 6 summarizes the issues that corresponded with ineffective debugging practices.

5 DISCUSSION

In this section, we connect our study findings to prior work focusing on knowledge, mental models, strategies, scaffolding approaches, and tactics. We then discuss the implications of our findings for teaching Python debugging in introductory courses.

Table 6. Issues Observed

Issue	**Examples**
Theme: Comprehension	
Fragile knowledge of language	All (4.1.1)
Poor searching for documented functions	Hugin (4.1.1)
Lack of familiarity with IDE features	Hugin (4.1.1)
Jumping to conclusions from single data point	Hugin (4.1.3)
Lack of methodological approach - jumping around ad hoc	Hugin (4.1.4)
Theme: Activities	
Introducing new bugs during code modification	Hugin and Jojo (4.2.3)
Making changes speculatively (trial and error)	All (4.2.3)
Not testing changes	JoJo and Hugin (4.2.3)
Lack of precision in speculation of cause	All (4.2.1)
Rewriting code that was not understood	JoJo and Hugin (4.2.3)
Theme: Workflow	
Lack of methodological approach - jumping around ad hoc	JoJo (4.3.2)
Lack of sustained activity	Hugin (4.3.3)

5.1 Domain Knowledge

Debugging is a process that requires several kinds of knowledge to be effective. Students in introductory programming courses are still learning the *domain knowledge* (i.e., the syntax and semantics of a programming language) which Jonassen and Hung [39] describe as a necessary prerequisite for troubleshooting. Similarly, Katz and Anderson [41] assumed that program comprehension is a precursor to debugging. However, it remains unclear how much domain knowledge is necessary. Although we were not specifically investigating student knowledge of Python, our observations allow us to infer aspects of their knowledge from their actions. All of the participants demonstrated an understanding of basic programming skills, but we also observed elements of fragile knowledge in all three cases. Despite this fragile knowledge, all three participants were able to engage in debugging activities to some extent. Our observations suggest that although some programming knowledge is necessary, and fragile understanding may reduce the effectiveness of debugging, strong *procedural knowledge* could compensate for gaps in domain knowledge.

5.2 Mental Models

The literature on novice programmers, while differing on the definition of mental models, is generally in agreement that novice programmer mental models are flawed and ineffectual [9, 43]. We refer the reader to a recent review of the literature on programmers' mental models for a comprehensive look at this area of research [34].

O'Dell [71] argued that all programs represent complex changes in state of a system over time and that programmers must therefore rely on mental models of the system during program development in order to be able to problem solve and reason about the behaviour of a program. These mental models are, in this context, considered to be approximations of the system. Our findings suggest that our participants lacked reliable mental models. However, there was some evidence that Django was able to build a reasonable mental model of the buggy program by connecting the program parts through an analysis of program flow and function input and output (see Section 4.1.4). In some cases, our participants believed that their flawed, incomplete, or inconsistent mental models were correct, which led them along a problem solving path that was incorrect and would not solve the bug. Occasionally, it appeared that their flawed mental models resulted in changes that introduced new bugs into the code. Even more problematic is when a mental model has been formed that is correct in a different context but does not fully align with a bug that students encounter during programming. O'Dell [71] highlighted the importance when debugging of recognising when a mental model needs to be discarded in order to develop a viable hypothesis — "Mental models can be used to intuit the causes of some bugs, but for the more difficult problems, relying on the mental model to describe the problem is exactly the wrong thing to do: the mental model is incorrect, which is why the bug happened in the first place. Throwing away the mental model is crucial to forming a sound hypothesis." (p. 8). Perhaps the fact that the students had yet to form reliable mental models and lacked the experience to know when their mental models were irrelevant to a bug explains why we observed the students focusing on lower-level tactics rather than hypothesis generation.

5.3 Procedural Knowledge of Debugging

Lewis [52] reported that the focus of student attention, rather than domain knowledge, was the main factor in the debugging success. Although our observations support the importance of student attention, we found that procedural knowledge about how to learn effectively is important for students with fragile knowledge to progress. This includes finding explanations of code elements in Python documentation, using the Internet to search for code examples, and executing code to verify understanding. As observed in Section 4.1, lack of domain knowledge prevented Hugin from identifying a bug in code that used dictionaries, but Djengo was able to overcome his lack of domain knowledge by looking up documentation and testing code fragments to build understanding. This procedural knowledge focuses on developing and verifying understanding of the domain, and is distinct from the procedural knowledge required to debug code when the domain is well understood. Our observations suggest that these two distinct kinds of procedural knowledge are required for novice programmers to effectively debug, and more emphasis should be placed on explicitly teaching such procedures in introductory courses.

We observed that when debugging a program, novices need to have procedural knowledge to effectively use the tools at their disposal. Both Djengo and JoJo attended to the organization of their work space (i.e., arrangement of windows and code), and were able to use the IDE provided to execute code. They also engaged in a modify-and-test cycle in which the code was executed after every change. These organizational processes that maximise information displayed may reduce cognitive load imposed by the need to remember information that is off-screen when switching windows. Similarly, executing code after a single change reduces the need to remember a sequence of changes and may also reduce cognitive overhead. This may contribute to effective debugging.

By comparison, Hugin made no attempt to organize the work space and frequently interacted inefficiently with the IDE. He made several changes to the code before executing it and appeared unfamiliar with the process to execute a function. This lack of familiarity with procedures used to effectively write and execute code in an IDE caused difficulty for Hugin when asked to debug code in that same environment.

5.4 Strategies

We did not observe clear evidence of high level strategies, either explicitly articulated or implicit in the observed student behaviours. The rainfall problem involved many functions and (relatively) complex flow of control that provided opportunity to demonstrate different strategies. Despite the opportunity for strategic debugging practices, we saw limited evidence that participants used different strategies on different problems. This is perhaps because the application of strategies requires experience beyond that typically acquired by introductory programmers. LaToza et al.

[48] show that even experienced developers who have acquired a range of debugging strategies are more successful when they are supported to focus on explicit debugging strategies.

5.5 Scaffolding

We found some evidence of general approaches that would be helpful in debugging – that is, approaches that would scaffold the debugging process. Djengo, who was the most effective at debugging, spent time understanding the purpose of the different functions in the rainfall variants and the relationship (i.e., the flow of control) between them. However, JoJo and Hugin did not familiarize themselves with the code and subsequently produced duplicated code that was found in other parts of the program. The lack of systematic program comprehension may be explained by difficulties in building mental models due to poor programming schema. The cognitive load imposed by modelling a program with many functions may exceed the capacity of some introductory students. This suggests that visual models may improve understanding of code structure and could potentially improve debugging processes.

Self-explanations are reported to be an effective practice for debugging [10, 55, 86]. Although the think-aloud data provided several examples of students developing their understanding of the code (see Section 4.1.4) and speculating about code output (see Section 4.2.1), we observed fewer examples of self-explanations extending to generating hypotheses about the cause of a bug — an important step in the heuristics for debugging described by Zeller [95]. When students speculated about the cause of a bug, the hypotheses lacked precision (see Section 4.2.1). This lack of precision in hypothesis generation may be a consequence of imprecise descriptions of the observed symptoms of the bug (see Section 4.1.3). For example, Djengo observed that "some of them [histogram bars] have more [Xs] than they should", but doesn't describe precisely that bars with a fractional part greater than 0.5 have 1 more X than they should.

Further, we observed that all of the participants could form hypotheses (see Section 4.1.3) using the difference between actual and expected output. This is consistent with findings reported in previous work involving pencil-and-paper exercises [91]. In this study, Djengo and JoJo were able to generalize from the cases in some problems despite a general lack of precision. However, they did not test their hypotheses by executing code. Instead, all students proceeded to attempt to localize the bug by reading the code.

Participants rarely took the time to consider multiple possible sources for bugs. They would focus on a single possible bug, only moving on to other hypotheses when that possibility was ruled out or they were unable to make progress with it due to other factors. This observation is consistent with the work of Fitzgerald et al. [29], in which only one of their participants mentioned considering alternatives.

5.6 Tactics

Our observations of student debugging practices provided numerous examples of what Metzger [66] describes as *tactical* behaviour. Participants demonstrated deliberate actions (such as printing the value of a variable) that were used to improve their understanding of code. These tactics might be considered the "building blocks" that underpin effective debugging practices. However, the tactics used by students appear to be a collection of techniques they have acquired in isolation and they do not appear to use them methodically.

Although students were taught code tracing, and were encouraged to use Online Python Tutor [33] as an aid for code tracing during their course instruction, it is interesting that we saw no evidence of formal code tracing on paper or the use of Online Python Tutor to provide a visualization of the machine state while executing code. Although we did see examples of code tracing, it was not systematic and students did not explicitly record memory state while tracing the program execution. It is possible that more explicit instruction and further practice with code tracing activities, as suggested in [93], may be needed for students to transfer their learning and deploy the techniques they learned in class more systematically.

5.7 Implications for Teaching

We observed no relationship between self-reported confidence and programming ability or debugging behaviour. The participants did not take a systematic approach to debugging and particularly to program comprehension. They made changes on a trial-and-error basis and often deleted changes without understanding whether they were correct or incorrect. Their activities were often in the form of tinkering. These findings are consistent with those reported by Ko et al. [47], who found that most students engaged in rapid cycles of editing and testing without deep understanding of the code, partly due to a sense that more systematic application of strategies slowed them down but also because they were not confident deploying strategies without help. This suggests that students have not acquired processes that support them to debug efficiently, and a greater focus on teaching debugging processes is needed. Because Python is interpreted and comes with a built-in interactive development environment, implementing a greater focus on testing small pieces of code may be easier for instructors and students.

Many of the issues that we observed derived from a lack of precision, lack of persistence, and lack of a methodical systematic approach. These are characteristics that relate to student meta-cognitive and self-regulation [58, 90]. Our findings align with the view advanced by Loksa and Ko [57] that there may be value in explicitly teaching self-regulation skills in introductory programming courses, but further work is needed to confirm the impact of such approaches.

Our study provides insight into the debugging approaches used by students and has revealed some possible avenues for teaching students to more effectively debug their programs, which complements suggestions arising from debugging research investigated in other contexts [47], such as explicit code tracing on paper [93]. Our findings suggest that teachers using Python in introductory courses may improve student debugging practices by:

- Eplicitly teaching processes for overcoming fragile knowledge (e.g., using documentation, testing code fragments to improve comprehension). The help function in Python makes access to documentation seamless in the built-in IDLE environment. As mentioned previously, the interactive nature of IDLE also encourages the testing of small pieces of code.
- Ensuring that students are familiar with standard development environments in courses that use specialised environments for assessment. The ease of using IDLE makes this suggestion simpler for both instructors and students.
- Teaching students to organize their windows and environments to maximize displayed information and reduce cognitive load.
- Encouraging students to generate *precise* descriptions of the symptoms when identifying the problem.
- Providing a visual model of programs with many functions to support program comprehension and flow of control.

6 LIMITATIONS

This case study recruited participants from a single institution, and reports a small number of cases in depth. All three participants identified as male, and all reported that they had previously been exposed to programming languages other than Python. The results are therefore highly contextualized and may not generalize broadly to other institutions, particularly if the instructional approach in those courses differs substantially. As with all case study research, the context is important when considering transferability and relevance of findings.

Students in this course mainly use an automated assessment environment (CodeRunner [56]) to develop their code for the practice exercises and assessments in the course. Those students who work exclusively within the web-based tool may not have developed familiarity with standard IDEs.

Additionally, the course instructional design focuses on students writing many small programs, an approach that has proven successful in other institutions [3]. However, it may be possible to debug code that involves only a small number of lines effectively by inspection. The CodeRunner system used by students to submit assessed exercises runs automated tests to provide correctness feedback. It is therefore possible that the teaching context supports students to debug by inspection and minimize testing.

The concept of "rubber-duck" debugging is used in industry [86] and education [10], and elements of think-aloud practices are evident in pair programming approaches that have been shown to be effective [69]. Therefore, it is possible that the think-aloud methodology used to collect information for this study has impacted on the debugging performance of students. However, debugging processes that involve explicit articulation of the problem are claimed to improve understanding and insight, suggesting that the performance of students in this study may be more effective than typical. Therefore, we expect that the issues and poor practices evident in the observations are perhaps underrepresented compared with typical student practices.

In this study, we asked students to debug programs authored by the researchers. In a previous study [92], we reported that students considered debugging code written by others to be different than code they had authored, perhaps resulting in different approaches.

7 CONCLUSION

In this work, we presented a case study involving an in-depth analysis, at a single point in time, of three novice programmers debugging code using a think-aloud observation protocol. These three students were selected to illustrate diversity of debugging performance: one relatively effective at debugging, one with average debugging practices, and one whose debugging was largely ineffective. This work contributes to the body of knowledge as one of the few articles that examines the debugging activities of novices without external guidance. We explored student debugging activities using a thematic analysis approach. This analysis resulted in the definition of a set of themes for future analysis of novice debuggers and we aligned the student debugging activities we observed with those reported in the literature.

Our observations revealed certain characteristics, such as self-awareness and perseverance, that led to student success during debugging. These are themes that have not historically been explored to any great extent in the context of debugging but are themes often explored in the broader program comprehension literature. More recently, there has been growing interest in explicitly addressing the interaction between emotional state, mindset, programming identity, and debugging strategies [19–21, 68, 80].

The current literature focuses more on debugging process rather than looking closely at the intertwining of knowledge, affective traits, and debugging ability for novices.

Several debugging tactics emerged in our analysis, including what we consider to be spontaneous tactics such as organisation of work space to see all relevant information. The student who exhibited this unguided tactic proved to be the most successful at debugging.

There was a notable connection between fragile programming knowledge, the inability to use the development environment, and a lack of success in debugging among the novice programmers observed in this study. Moreover, some personal characteristics had real impact on the participants' debugging ability. The most successful of the three debuggers demonstrated perseverance and a more methodical approach to debugging than the other two in this study. Our observations also support the notion that procedural knowledge and strategies for filling in knowledge gaps are a prerequisite for debugging success. However, we would note that this is only true if the student is persistent and perseveres when faced with debugging roadblocks.

We found that there was some link between debugging tactics used and complexity of the debugging task. The most successful participant tended towards more of a systematic approach as the problem became more complex. As a result, we suggest that encouraging effective debugging practices requires not only time but also exposure to larger debugging problems for which a systematic approach is required to successfully eliminate the bugs.

All three students had some effective practices and tactics. This is encouraging because it demonstrates that students have taken the first steps towards effective debugging practices despite the tactics being applied sporadically. We anticipate that pedagogies building on these tactics may be effective for novice programmers.

Finally, there is a modest body of work comparing experts' and novice programmers' debugging processes. However, our results suggest that some debugging practices that should be taught to students are not necessarily practices reported in the literature on expert debugging. For example, we feel that there is value in explicitly teaching students to organize their work environment (windows) to maximize information display. However, experts typically use professional development environments that assist in this organization. Therefore, work space knowledge is not typically reported in literature on expert debuggers. Further research explicitly articulating implicit expert practices that would benefit novice debuggers is warranted.

7.1 Future Work

An interesting extension of this work would be to map a timeline for each participant that shows how much time the novice debuggers spend on different debugging activities. This would help us to identify learning trajectories, the barriers that impact on student progression, and the effort spent on particular activities. Understanding these barriers and the ordered steps that students take when debugging has the potential to lead to more effective teaching practices.

Future researchers should consider pushing the boundary of unstructured and unguided studies and observe novice debuggers debugging their own code on fairly substantial novice programming tasks containing their own naturally occurring bugs. Such studies might include in-depth investigations into the role of self-regulation in the development of effective novice debuggers. Another interesting area for study might be further investigation into the roles of metacognition, grit, and affective learning as a predictors of debugging success.

Another aspect worthy of study is pair-debugging and whether it is an effective learning experience for building debugging skills. This introduces the potential to explore novice debugging in the context of theories related to socially shared regulation and co-regulation.

Finally, our study revealed potentially complex interactions between domain and procedural knowledge (e.g., using Google to look up missing elements of domain knowledge). A study to explore these interactions in more depth may provide valuable insights.

APPENDIX

A TEST CASES FOR DEBUGGING EXERCISES

A.1 Warm-up Exercise

The participants were given two separate sample runs of the warm-up function, provided here.

```
data = 'supercalifragilisticexpialidocious'

substring(data, 0, 4)
```

```
Expected output:
'super'

Actual output:
'supe'

data = 'supercalifragilisticexpialidocious'

substring(data, 5, 19)

Expected output:
'califragilistic'

Actual output:
'califragilisti'
```

A.2 Dictionary

The first test case shown to the participants for the dictionary problem does not reveal any bugs in the code, as it works perfectly. The second test case provided reveals the issue with capitalization, recording differing votes for capitalized versus lowercase strings. The third test case was provided to reveal the problem with the check in the if statement. Here, because the wrong dictionary is being checked, the function raises an exception since 'Amber' is not a key in the dictionary d3 provided as a parameter.

```
d1 = {'Gertrude': 'Simone', 'Prudence': 'Andre', 'Amber': 'Chad'}

record_votes(d1)

Expected output:
{'Simone': 1, 'Andre': 1, 'Chad': 1}

Actual output:
{'Simone': 1, 'Andre': 1, 'Chad': 1}

d2 = {'Gertrude': 'simone', 'Prudence': 'Andre', 'Amber': 'SIMONE'}

record_votes(d2)

Expected output:
{'Simone': 2, 'Andre': 1}

Actual output:
{'simone': 1, 'Andre': 1, 'SIMONE': 1}

d3 = {'Gertrude': 'Amber', 'Prudence': 'Andre', 'Amber': 'Erin',
      'Erin': 'Amber'}

record_votes(d3)

Expected output:
{'Amber': 2, 'Andre': 1, 'Erin': 1}

Actual output:
Traceback (most recent call last):
  File "<pyshell#7>", line ..., in <module>
    record_votes(d3)
  File ..., line ..., in record_votes
    votes[vote_dict[k]] += 1
KeyError: 'Amber'
```

A.3 Rainfall A

For the Rainfall A problem, participants were provided with a single test case and resulting output.

```
raw data file contents:
10 10 6 -20 25 13 12 10 10 20 -32
5 6 2 -27 3 1 3 0 0 0 2
0 0 34 0 0 0 2 1
20 25 -34 26 27 28 10 15
20 32 2 10 2
56 43
12 19 5 0 24 0 52 43 10
8 2 -10 -30 -1 1

10
```

```
Expected output:
----- Monthly Rainfall -----
J: XXXXXXXXXXXXX (12.9)
F: XX (2.2)
M: XXXXX (4.6)
A: XXXXXXXXXXXXXXXXXXXXXX (21.6)
M: XXXXXXXXXXXXX (13.2)
J: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (49.5)
J: XXXXXXXXXXXXXXXXXX (18.3)
A: XXXX (3.7)
S:  (0.0)
O: XXXXXXXXXX (10.0)
N:  (0.0)
D:  (0.0)
----- Summary Statistics -----
Monthly high: 49.5
Monthly low: 0.0
Monthly median: 7.3
Monthly mean: 11.3


Actual output:
----- Monthly Rainfall -----
J: XXXXXX (5.8)
F:  (-0.5)
M: XXXXX (4.6)
A: XXXXXXXXXXXXXXX (14.6)
M: XXXXXXXXXXXXX (13.2)
J: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (49.5)
J: XXXXXXXXXXXXXXXXXX (18.3)
A:  (-5.0)
S:  (0.0)
O: XXXXXXXXXX (10.0)
N:  (0.0)
D:  (0.0)
----- Summary Statistics -----
Monthly high: 49.5
Monthly low: -5.0
Monthly median: 5.2
Monthly mean: 9.2
```

A.4 Rainfall B

For the Rainfall B problem, participants were also provided with a single test case and resulting output.

```
Expected output:
----- Monthly Rainfall -----
J: XXXXXXXXXXXXX (12.9)
F: XX (2.2)
M: XXXXX (4.6)
A: XXXXXXXXXXXXXXXXXXXXXX (21.6)
M: XXXXXXXXXXXXX (13.2)
J: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (49.5)
J: XXXXXXXXXXXXXXXXXX (18.3)
A: XXXX (3.7)
S:  (0.0)
O: XXXXXXXXXX (10.0)
N:  (0.0)
D:  (0.0)
----- Summary Statistics -----
Monthly high: 49.5
Monthly low: 0.0
Monthly median: 7.3
Monthly mean: 11.3

Actual output:
----- Monthly Rainfall -----
J: XXXXXXXXXXXX (12.9)
F: XX (2.2)
M: XXXX (4.6)
A: XXXXXXXXXXXXXXXXXXXXX (21.6)
M: XXXXXXXXXXXXX (13.2)
J: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (49.5)
J: XXXXXXXXXXXXXXXXXX (18.3)
A: XXX (3.7)
S:  (0.0)
O: XXXXXXXXXX (10.0)
N:  (0.0)
D:  (0.0)
```

```
----- Summary Statistics -----
Monthly high: 49.5
Monthly low: 0.0
Monthly median: 7.3
Monthly mean: 11.3
```

A.5 Rainfall C

Like the other rainfall problems, the participants were given a single test case and resulting output.

```
Expected output:
----- Monthly Rainfall -----
J: XXXXXXXXXXXXX (12.9)
F: XX (2.2)
M: XXXXX (4.6)
A: XXXXXXXXXXXXXXXXXXXXXX (21.6)
M: XXXXXXXXXXXXX (13.2)
J: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (49.5)
J: XXXXXXXXXXXXXXXXXX (18.3)
A: XXXX (3.7)
S:  (0.0)
O: XXXXXXXXXX (10.0)
N:  (0.0)
D:  (0.0)
----- Summary Statistics -----
Monthly high: 49.5
Monthly low: 0.0
Monthly median: 7.3
Monthly mean: 11.3

Actual output:
----- Monthly Rainfall -----
J:  (0.0)
F:  (0.0)
M:  (0.0)
A:  (0.0)
M:  (0.0)
J:  (0.0)
J:  (0.0)
A:  (0.0)
S:  (0.0)
O:  (0.0)
N:  (0.0)
D:  (0.0)
----- Summary Statistics -----
Monthly high: 0.0
Monthly low: 0.0
Monthly median: 0.0
Monthly mean: 0.0
```

Because of the number of bugs in the code, it was not expected that this test case would be sufficient for the participants to complete the debugging. The debugging of Rainfall C would require several interactive rounds of debugging and execution of the corrected code.

ACKNOWLEDGMENTS

This study was approved by the University of Auckland Human Participants Ethics Committee (UAHPEC Ref 024038).

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'05)*. ACM, New York, NY, 84–88. https://doi.org/10.1145/1067445.1067472

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2007. The impact of improving debugging skill on programming ability. *Innovation in Teaching and Learning in Information and Computer Sciences* 6, 4 (2007), 72–87. https://doi.org/10.11120/ital.2007.06040072

[3] Joe Michael Allen, Frank Vahid, Alex Edgcomb, Kelly Downey, and Kris Miller. 2019. An analysis of using many small programs in CS1. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 585–591. https://doi.org/10.1145/3287324.3287466

[4] Amjad Altadmri and Neil C. C. Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)*. ACM, New York, NY, 522–527. https://doi.org/10.1145/2676723.2677258

[5] Nabeel Alzahrani and Frank Vahid. 2021. Common logic errors for programming learners: A three-decade literature survey. In *ASEE Virtual Annual Conference Content Access*. ASEE Conferences, Virtual Conference. https://peer.asee.org/36814.

[6] Brett Becker and Catherine Mooney. 2016. Categorizing compiler error messages with principal component analysis. In *12th China-Europe International Symposium on Software Engineering Education (CEISEE'16)*. https://researchrepository.ucd.ie/handle/10197/7889.

[7] Susan Bergin, Ronan Reilly, and Desmond Traynor. 2005. Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the 1st International Workshop on Computing Education Research (ICER'05)*. ACM, New York, NY, 81–86. https://doi.org/10.1145/1089786.1089794

[8] David B. Bisant and Lowell Groninger. 1993. Cognitive processes in software fault detection: A review and synthesis. *International Journal of Human–Computer Interaction* 5, 2 (1993), 189–206. https://doi.org/10.1080/10447319309526064

[9] Richard Bornat, Saeed Dehnadi, and Simon. 2008. Mental models, consistency and programming aptitude. In *Proceedings of the 10th Conference on Australasian Computing Education - Volume 78 (ACE'08)*. Australian Computer Society, Inc., AUS, 53–61.

[10] A. C. Bradley. 2020. Rubber ducks and double labs: Teaching debugging strategies in a geoscience class. In *AGU Fall Meeting Abstracts*, Vol. 2020. Article ED011-04, ED011-04 pages.

[11] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa arXiv:https://www.tandfonline.com/doi/pdf/10.1191/1478088706qp063oa.

[12] Virginia Braun and Victoria Clarke. 2022. Conceptual and design thinking for thematic analysis. *Qualitative Psychology* 9, 1 (2022), 3–26. https://doi.org/10.1037/qup0000196

[13] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and validating Java misconceptions toward a CS1 concept inventory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'19)*. ACM, New York, NY, 23–29. https://doi.org/10.1145/3304221.3319771

[14] Elizabeth Carter. 2015. Its debug: Practical results. *Journal of Computing Sciences in Colleges* 30, 3 (jan 2015), 9–15.

[15] Thomas P. Cavaiani. 1989. Cognitive style and diagnostic skills of student programmers. *Journal of Research on Computing in Education* 21, 4 (1989), 411–420.

[16] Mei-Wen Chen, Cheng-Chih Wu, and Yu-Tzu Lin. 2013. Novices' debugging behaviors in VB programming. In *2013 Learning and Teaching in Computing and Engineering*. 25–30. https://doi.org/10.1109/LaTiCE.2013.38

[17] Ryan Chmiel and Michael C. Loui. 2004. Debugging: From novice to expert. *ACM SIGCSE Bulletin* 36, 1 (2004), 17–21.

[18] Marilyn Cochran-Smith and Susan Lytle. 1993. *Inside/outside: Teacher Research and Knowledge*. Teachers College Press, New York.

[19] Maggie Dahn and David DeLiema. 2020. Dynamics of emotion, problem solving, and identity: Portraits of three girl coders. *Computer Science Education* 30, 3 (2020), 362–389. https://doi.org/10.1080/08993408.2020.1805286

[20] David DeLiema, Maggie Dahn, Virginia J. Flood, Dor Abrahamson, Noel Enyedy, and Francis Steen. 2020. Debugging as a context for collaborative reflection on problem-solving processes. In *Deeper Learning, Communicative Competence, and Critical Thinking: Innovative, Research-Based Strategies for Development in 21st Century Classrooms*, E. Manolo (Ed.). Routledge, 209–228.

[21] David DeLiema, Maggie Dahn, Virginia J. Flood, Ana Asuncion, Dor Abrahamson, Noel Enyedy, and Francis Steen. 2019. Debugging as a context for fostering reflection on critical thinking and emotion. In *Deeper Learning, Dialogic Learning, and Critical Thinking*. Routledge, 1–20. https://doi.org/10.4324/9780429323058

[22] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. ACM, New York, NY, 75–80. https://doi.org/10.1145/2325296.2325318

[23] Chuntao Du. 2009. Empirical study on college students' debugging abilities in computer programming. In *2009 1st International Conference on Information Science and Engineering*. 3319–3322. https://doi.org/10.1109/ICISE.2009.550

[24] Bendict DuBoulay. 1989. Some difficulties of learning to program. In *Studying the Novice Programmer*, E. Soloway and J. C. Spohrer (Eds.). Number 14. Lawrence Erlbaum Associates, 283–299. http://www.sussex.ac.uk/Users/bend/papers/diffsofprogramming.pdf.

[25] Mireille Ducasse and A.-M. Emde. 1988. A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings of the 10th International Conference on Software Engineering*. IEEE Computer Society Press, 162–171.

[26] Kiran L. N. Eranki and Kannan M. Moudgalya. 2014. Application of program slicing technique to improve novice programming competency in spoken tutorial workshops. In *IEEE 6th International Conference on Technology for Education (T4E'14)*. IEEE, 32–35.

[27] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference (ACE'18).* ACM, New York, NY, 83–89. https://doi.org/10.1145/3160489.3160493

[28] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116. https://doi.org/10.1080/08993400802114508 arXiv:https://doi.org/10.1080/08993400802114508.

[29] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2010. Debugging from the student perspective. *IEEE Transactions on Education* 53, 3 (Aug 2010), 390–396. https://doi.org/10.1109/TE.2009.2025266

[30] CC2020 Task Force. 2020. *Computing Curricula 2020: Paradigms for Global Computing Education.* ACM, New York, NY.

[31] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Identifying important and difficult concepts in introductory computing courses using a delphi process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education.* 256–260.

[32] Valentina Grigoreanu, James Brundage, Eric Bahna, Margaret Burnett, Paul ElRif, and Jeffrey Snover. 2009. Males' and Females' script debugging strategies. In *End-User Development*, Volkmar Pipek, Mary Beth Rosson, Boris de Ruyter, and Volker Wulf (Eds.). Springer, Berlin, 205–224.

[33] Philip J. Guo. 2013. Online Python tutor: Embeddable web-based program visualization for Cs education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13).* ACM, New York, NY, 579–584. https://doi.org/10.1145/2445196.2445368

[34] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. 2022. Synthesizing Research on Programmers' Mental Models of Programs, Tasks and Concepts – a Systematic Literature Review. (2022). https://doi.org/10.48550/ARXIV.2212.07763

[35] Ting-Yun Hou, Yu-Tzu Lin, Yu-Chih Lin, Chia-Hu Chang, and Miao-Hsuan Yen. 2013. Exploring the gender effect on cognitive processes in program debugging based on eye-movement analysis. In *Proceedings of the 5th International Conference on Computer Supported Education.* ResearchGate, 16–21.

[36] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bulletin* 35, 1 (Jan 2003), 153–156. https://doi.org/10.1145/792548.611956

[37] Gayithri Jayathirtha, Deborah Fields, and Yasmin Kafai. 2020. Pair debugging of electronic textiles projects: Analyzing think-aloud protocols for high school students' strategies and practices while problem solving. In *ICLS 2020: The 14th International Conference of the Learning Sciences. International Society of the Learning Sciences (ISLS'20).* https://repository.isls.org//handle/1/6292.

[38] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* ACM, New York, NY.

[39] David H. Jonassen and Woei Hung. 2006. Learning to troubleshoot: A new theory-based design architecture. *Educational Psychology Review* 18 (2006), 77–114. https://doi.org/10.1007/s10648-006-9001-8

[40] Yasmin B. Kafai, David DeLiema, Deborah A. Fields, Gary Lewandowski, and Colleen Lewis. 2019. Rethinking debugging as productive failure for CS education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19).* ACM, New York, NY, 169–170. https://doi.org/10.1145/3287324.3287333

[41] Irvin R. Katz and John R. Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human–Computer Interaction* 3, 4 (Dec. 1987), 351–399. https://doi.org/10.1207/s15327051hci0304_2

[42] Cazembe Kennedy and Eileen T. Kraemer. 2018. What are they thinking? Eliciting student reasoning about troublesome concepts in introductory computer science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling'18).* ACM, New York, NY, Article 7, 10 pages. https://doi.org/10.1145/3279720.3279728

[43] Claudius M. Kessler and John R. Anderson. 1986. Learning flow of control: Recursive and iterative procedures. *Human–Computer Interaction* 2, 2 (Jun 1986), 135–166. https://doi.org/10.1207/s15327051hci0202_2

[44] Louise H. Kidder and Michelle Fine. 1987. Qualitative and quantitative methods: When stories converge. *New Directions for Program Evaluation* 1987, 35 (1987), 57–75. https://doi.org/10.1002/ev.1459 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/ev.1459.

[45] E. L. N. Kiran and K. M. Moudgalya. 2015. Evaluation of programming competency using student error patterns. In *2015 International Conference on Learning and Teaching in Computing and Engineering.* 34–41. https://doi.org/10.1109/LaTiCE.2015.16

[46] Amy Ko. 2020. The CS-Ed Podcast. (2020). Retrieved March 17, 2020 from https://sites.duke.edu/csedpodcast/2020/01/07/episode-3-amy/.

[47] Amy J. Ko, Thomas D. LaToza, Stephen Hull, Ellen A. Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching explicit programming strategies to adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 469–475. https://doi.org/10.1145/3287324.3287371

[48] Thomas D. LaToza, Maryam Arab, Dastyni Loksa, and Amy J. Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25, 4 (2020), 2416–2449.

[49] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2013), 197–215. https://doi.org/10.1109/TSE.2010.111

[50] Greg C. Lee and Jackie C. Wu. 1999. Debug it: A debugging practicing system. *Computers & Education* 32, 2 (1999), 165–179.

[51] Michael Jong Lee. 2015. *Teaching and engaging with debugging puzzles.* Ph.D. Dissertation. University of Washington.

[52] Colleen M. Lewis. 2012. The importance of students' attention to program state: A case study of debugging behavior. In *Proceedings of the 9th Annual International Conference on International Computing Education Research (ICER'12)*. ACM, New York, NY, 127–134. https://doi.org/10.1145/2361276.2361301

[53] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a framework for teaching debugging. In *Proceedings of the 21st Australasian Computing Education Conference (ACE'19)*. ACM, New York, NY, 79–86.

[54] Y. Lin, C. Wu, T. Hou, Y. Lin, F. Yang, and C. Chang. 2016. Tracking students' cognitive processes during program debugging–an eye-movement approach. *IEEE Transactions on Education* 59, 3 (Aug 2016), 175–186. https://doi.org/10.1109/TE.2015.2487341

[55] Zhongxiu Liu, Rui Zhi, Andrew Hicks, and Tiffany Barnes. 2017. Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education* 27, 1 (2017), 1–29. https://doi.org/10.1080/08993408.2017.1308651 arXiv:https://doi.org/10.1080/08993408.2017.1308651.

[56] Richard Lobb and Jenny Harlow. 2016. Coderunner: A tool for assessing computer programming skills. *ACM Inroads* 7, 1 (Feb 2016), 47–51. https://doi.org/10.1145/2810041

[57] Dastyni Loksa and Amy J. Ko. 2016. The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER'16)*. ACM, New York, NY, 83–91. https://doi.org/10.1145/2960310.2960334

[58] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2021. Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education* (Dec 2021). https://doi.org/10.1145/3487050 Just Accepted.

[59] Dastyni Loksa, Benjamin Xie, Harrison Kwik, and Amy J. Ko. 2020. Investigating novices' in situ reflections on their programming process. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. ACM, New York, NY, 149–155. https://doi.org/10.1145/3328778.3366846

[60] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. 2018. Ladebug: An online tool to help novice programmers improve their debugging skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'18)*. ACM, New York, NY, 159–164. https://doi.org/10.1145/3197091.3197098

[61] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the viability of mental models held by novice programmers. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07)*. ACM, New York, NY, 499–503. https://doi.org/10.1145/1227310.1227481

[62] Julia M. Markel and Philip J. Guo. 2021. *Inside the Mind of a CS Undergraduate TA: A Firsthand Account of Undergraduate Peer Tutoring in Computer Labs.* ACM, New York, NY, 502–508. https://doi.org/10.1145/3408877.3432533

[63] Davin McCall. 2016. *Novice Programmer Errors - Analysis and Diagnostics.* Ph.D. Dissertation. University of Kent. https://kar.kent.ac.uk/61340/.

[64] Davin McCall and Michael Kölling. 2019. A new look at novice programmer errors. *ACM Transactions on Computing Education* 19, 4, Article 38 (jul 2019), 30 pages. https://doi.org/10.1145/3335814

[65] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[66] Robert Charles Metzger. 2004. *Debugging by Thinking: A Multidisciplinary Approach.* Elsevier Digital Press.

[67] Michael A. Miljanovic and Jeremy S. Bradbury. 2017. RoboBUG: A serious game for learning debugging techniques. In *Proceedings of the 2017 ACM Conference on International Computing Education Research.* ACM, 93–100.

[68] L. Morales-Navarro, D. A. Fields, and Y. B. Kafai. 2021. Growing mindsets: Debugging by design to promote students' growth mindset practices in computer science class. In *Proceedings of the 15th International Conference of the Learning Sciences (ICLS'21)*, E. de Vries, Y. Hod, and J. Ahn (Eds.). https://par.nsf.gov/biblio/10309425.

[69] Laurie Murphy, Sue Fitzgerald, Brian Hanks, and Renée McCauley. 2010. Pair debugging: A transactive discourse analysis. In *Proceedings of the 6th International Workshop on Computing Education Research (ICER'10)*. ACM, New York, NY, 51–58. https://doi.org/10.1145/1839594.1839604

[70] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: The good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08)*. ACM, New York, NY, 163–167. https://doi.org/10.1145/1352135.1352191

[71] Devon H. O'Dell. 2017. The debugging mindset. *Queue* 15, 1 (Feb. 2017), 71–90. https://doi.org/10.1145/3055301.3068754

[72] D. N. Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *1st Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. 213–229.

[73] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER'18)*. ACM, New York, NY, 41–50. https://doi.org/10.1145/3230977.3230981

[74] David Pritchard. 2015. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'15)*. ACM, New York, NY, 1–8. https://doi.org/10.1145/2846680.2846681

[75] S. Rao and V. Kumar. 2008. A theory-centric real-time assessment of programming. In *2008 8th IEEE International Conference on Advanced Learning Technologies*. 139–143. https://doi.org/10.1109/ICALT.2008.288

[76] Elizabeth Susan Rezel. 2003. *The Effect of Training Subjects in Self-explanation Strategies on Problem Solving Success in Computer Programming*. Marquette University.

[77] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13 (2003), 137–172.

[78] Joseph R. Ruthruff, Shrinu Prabhakararao, James Reichwein, Curtis Cook, Eugene Creswick, and Margaret Burnett. 2005. Interactive, visual fault localization support for end-user programmers. *Journal of Visual Languages & Computing* 16, 1-2 (2005), 3–40.

[79] Alma Schaafstal, Jan Maarten Schraagen, and Marcel van Berl. 2000. Cognitive task analysis and innovation of training: The case of structured troubleshooting. *Human Factors* 42, 1 (2000), 75–86. https://doi.org/10.1518/001872000779656570 arXiv:https://doi.org/10.1518/001872000779656570. PMID: 10917147.

[80] Michael J. Scott and Gheorghita Ghinea. 2014. On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education* 57, 3 (2014), 169–174. https://doi.org/10.1109/TE.2013.2288700

[81] Leonardo Silva, Antonio Mendes, Anabela Gomes, Gabriel Fortes, Chan Tong Lam, and Calana Chan. 2021. Exploring the association between self-regulation of learning and programming learning: A multinational investigation. In *2021 IEEE Frontiers in Education Conference (FIE'21)*. 1–8. https://doi.org/10.1109/FIE49875.2021.9637438

[82] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. 2018. Language choice in introductory programming courses at Australasian and UK universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*. ACM, New York, NY, 852–857. https://doi.org/10.1145/3159450.3159547

[83] Rebecca Smith and Scott Rixner. 2019. The error landscape: Characterizing the mistakes of novice programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 538–544. https://doi.org/10.1145/3287324.3287394

[84] E. Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594

[85] Neeraja Subrahmaniyan, Cory Kissinger, Kyle Rector, Derek Inman, Jared Kaplan, Laura Beckwith, and Margaret Burnett. 2007. Explaining debugging strategies to end-user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2007 (VL/HCC'07)*. IEEE, 127–136.

[86] David Thomas and Andrew Hunt. 2019. *The Pragmatic Programmer: Your Journey to Mastery*. Addison-Wesley Professional.

[87] Maarten van Someren. 1994. The think aloud method: A practical guide to modelling cognitive processes.

[88] Ashok Kumar Veerasamy, Daryl D'Souza, and Mikko-Jussi Laakso. 2016. Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology Systems* 45, 1 (2016), 50–73. https://doi.org/10.1177/0047239515627263 arXiv:https://doi.org/10.1177/0047239515627263.

[89] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. 2020. G4D — A treasure hunt game for novice programmers to learn debugging. *Smart Learning Environments* 7, 1 (2020), 21.

[90] Ethan Wert, Jeremy Grifski, Sijia Luo, and Zahra Atiq. 2021. A multi-modal investigation of self-regulation strategies adopted by first-year engineering students during programming tasks. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER'21)*. ACM, New York, NY, USA, 446–447. https://doi.org/10.1145/3446871.3469795

[91] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Analysis of a process for introductory debugging. In *Australasian Computing Education Conference (ACE'21)*. ACM, New York, NY, 11–20. https://doi.org/10.1145/3441636.3442300

[92] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice reflections on debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE'21)*. ACM, New York, NY, 73–79. https://doi.org/10.1145/3408877.3432374

[93] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*. ACM, New York, NY, 344–349. https://doi.org/10.1145/3159450.3159527

[94] Ching-Zon Yen, Ping-Huang Wu, and Ching-Fang Lin. 2012. Analysis of experts' and novices' thinking process in program debugging. In *Engaging Learners Through Emerging Technologies*, Kam Cheong Li, Fu Lee Wang, Kin Sun Yuen, Simon K. S. Cheung, and Reggie Kwan (Eds.). Springer, Berlin, 122–134.

[95] Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier.

Received 31 May 2022; revised 4 March 2023; accepted 10 March 2023