Investigating the Utility of Notional Machine Instruction in an Introductory Programming Lesson

by

Veronica Chiarelli

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Cognitive Science

Carleton University
Ottawa, Ontario

© 2022
Veronica Chiarelli
Abstract

Learning to program involves understanding how the computer executes programs. Code tracing is a simulation of the steps the computer takes to execute a program. The notional machine is an abstract representation of this process. Students form mental models of the notional machine when learning to code trace, but these mental models can be inaccurate or contain misconceptions. We experimentally investigated the effect on learning of using an explicit notional machine or not in a code-tracing lesson for novice programmers ($N = 48$). We created two versions of a tutoring system, one with a notional machine and one without, using the Cognitive Tutor Authoring Tools framework. The tutors included video lessons and self-explanation prompts to encourage participant engagement. We measured learning as the difference in scores from pretest to posttest, adjusted by prior knowledge. Learning increased overall, but there was no significant difference in learning between groups.
Acknowledgements

I would like to dedicate this thesis to my grandfather Frantisek (Frank) Fiala (1935-2021) who was a founding faculty member of Carleton University’s School of Computer Science. He has inspired my love for learning through his example and gentle encouragement, and I am so grateful for that.

I would also like to thank my friends and family for constantly cheering me on, especially my mother for her enthusiasm and willingness to hear me talk for hours about the details of my research.

Thank you to my committee for your time, input and ideas.

Finally, thank you to Dr. Kasia Muldner for making this possible. I greatly appreciate all of the time on discussions and feedback that went into this work. Thank you for being such a kind and helpful supervisor.
Contents

Abstract i

Acknowledgements ii

List of Tables vi

List of Figures viii

1 Introduction 1

2 Related Work 4

2.1 Code Tracing .................................................. 4

2.2 Learning to Code Trace with Explicit and Implicit Notional Machine Instruction .............................................. 7

2.3 Computer Tools to Support Novice Programmers .......................................................... 14

2.4 Self-Explanation .................................................. 17

2.5 Summary .................................................. 19

3 Pilot 21

3.1 Participants .................................................. 21
B Study Materials

B.1 Consent Form .................................................. 109
B.2 Demographics Questionnaire .............................. 113
B.3 Pretest ............................................................ 113
B.4 Prompts in the Tutor ......................................... 118
B.5 Posttest ........................................................... 120
List of Tables

3.1 Self-Explanation Prompts Used in the Lesson. ........................... 22
3.2 Interview Questions. .............................................................. 25
3.3 Self-Explanation Categories and Sample Participant Self-Explanations,
    with Notional Machine Intrusions in Bold ............................... 28
3.4 Self-Explanations in Each Category, per Participant ................... 29
3.5 Number of Correct Answers (Out of 8 Questions) on Pretest and
    Posttest and Learning (Increase in Scores from Pretest to Posttest),
    per Participant ................................................................. 33
4.1 Instructional Video Lengths for Experimental and Control Versions. . 42
4.2 Sequence of Tutor Screens. ...................................................... 49
5.1 Descriptives for the Pretest and Posttest, per Condition ............... 55
5.2 Normalized Gain, per Condition ............................................. 57
5.3 Descriptives for Scores on Easy and Hard Questions, per Condition. 58
5.4 Descriptives for Normalized Gain Scores for Easy and Hard Questions,
    per Condition ................................................................. 59
5.5 Example of Oral Response to Question 7 Demonstrating that the Control Participant has Formed an Accurate Mental Model of the Notional Machine. .......................... 65

5.6 Example of Oral Response to Question 7 Demonstrating that the Experimental Participant has Formed an Accurate Mental Model of the Notional Machine. .......................... 66

5.7 Examples of Oral Responses to Question 7 Demonstrating Misconceptions in Mental Models Formed by Some Participants. .......................... 67
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A Python Program (Left) and Its Code Trace (Right).</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Notional Machine and Its Components.</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>State of the Notional Machine After Executing The First Four Lines of a Seven-Line Program Involving a While Loop.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>A While Loop Program to Code Trace on the Posttest.</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Program to be Traced for Pretest Question 4, Targeting the Common Misconception that the Computer Understands the Purpose of a Given Program.</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Notional Machine Used in the Experimental Group’s Programming Lesson.</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>A Two-Line Python Program.</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>State of the Notional Machine Following the Execution of Line 1 in the Program, with Changes Resulting from Line 1 Highlighted in Yellow.</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>State of the Notional Machine Following the Execution of Line 2 in the Program, with the Changes Resulting from Line 2 Highlighted in Yellow.</td>
<td>40</td>
</tr>
<tr>
<td>4.6</td>
<td>An Example of Instructional Video Layout.</td>
<td>41</td>
</tr>
<tr>
<td>4.7</td>
<td>A Simple While Loop Program.</td>
<td>41</td>
</tr>
</tbody>
</table>
4.8 An Example of a Control Video Demonstrating Standard Code Tracing
with a Code-Trace Table (Orange Table on the Right Side of the Screen). 42
4.9 A Program with a While Loop, Used in a Prompt. . . . . . . . . . . . 43
4.10 An Instruction Screen in the Tutor. . . . . . . . . . . . . . . . . . . . 44
4.11 A Message from the Tutor Confirming that the User is Done After
They Have Clicked the Done Button. . . . . . . . . . . . . . . . . . . . . . . 45
4.12 A Video Screen in the Tutor. . . . . . . . . . . . . . . . . . . . . . . . 46
4.13 A Prompt Screen in the Tutor. . . . . . . . . . . . . . . . . . . . . . . 47
4.14 A Prompt Screen in the Tutor with Response Boxes Next to Each Line
of the Program Being Code Traced. . . . . . . . . . . . . . . . . . . . . . . . . 48
4.15 Left: HTML Interface, Right: Corresponding Behaviour Graph. . . . 51
5.1 Mean Pretest Scores (%) and Mean Posttest Scores (%), by Condition. 55
5.2 Mean Normalized Gain (%), by Condition. . . . . . . . . . . . . . . . 57
5.3 Prompt 2 in the Tutor. . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.4 Prompt 4 in the Tutor. . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.5 Python Program to be Code Traced for Posttest Question 7. . . . . . 64
Chapter 1

Introduction

Code tracing is the simulation of the steps the computer takes to execute a program (Cunningham et al., 2017). This is a fundamental skill necessary for learning to program (Xie et al., 2019). To code trace, students need an appropriate mental model of the notional machine (Du Boulay, 1986), which is an abstraction of a computer executing a program. Students develop notional machine mental models from code-tracing activities, but commonly these mental models are inaccurate and/or incomplete (Sorva, 2013).

Explicit instruction about the notional machine has been proposed to support mental model formation for novice programmers (Krishnamurthi and Fisler, 2019; Lowe, 2018; Robins, 2019; Sorva, 2013). However, to date this proposal remains relatively untested. Mayer (1976) developed a notional machine with abstractions of computer components necessary for code-tracing simple programs: memory (to keep track of variables and their values), input, output, and program and program pointer (to monitor the flow of execution). Mayer (1975), Mayer (1976) and Bayman and Mayer (1988) reported learning benefits for novice programmers learning to code trace
from lessons including this notional machine, but that work dates back over 30 years.

The present thesis investigates the utility of teaching code tracing with a notional machine versus without one, extending prior research to a computer tutor with video lessons and self-explanation prompts. We hypothesized the experimental group (learning to code trace with a notional machine) to benefit from the additional mental model formation support provided by the notional machine and therefore to learn more than the control group (learning to code trace without a notional machine).

To test the effect of notional machine instruction, we designed two versions of an introductory programming lesson that included instructional videos on basic programming concepts in the language Python. We based our notional machine model on Mayer (1976)'s notional machine and refined it according to feedback from our pilot study. The instructional videos in the experimental version used the refined notional machine when demonstrating code-tracing activities. In contrast, the videos in the control version used a table to keep track of variable values but did not explicitly represent all computer components and their interaction during code traces. To encourage engagement with the materials, both lesson versions included identical prompts to self-explain key concepts. Each lesson was embedded in a computer tutor we implemented using the Cognitive Tutor Authoring Tools framework. The advantage of this approach is that it allows us to specify tutor behavior (e.g., list acceptable interactions or responses in the tutor) and automatically logs student actions in the system (Aleven et al., 2009, 2016).

We compared the two lesson versions using a between-subjects experiment with novice programmers as participants ($N = 48$). Participants took a pretest and a posttest. Learning was measured as differences between pretest and posttest scores, adjusted by prior knowledge. We compared learning in both conditions overall and
based on question difficulty.

We begin by presenting related work. We then describe our pilot followed by our study’s lesson and tutoring system. Finally, we report on results and discuss findings.
Chapter 2

Related Work

2.1 Code Tracing

Learning to program involves learning how to use programming languages, understand algorithms (i.e., high-level steps to accomplish a goal), write programs using algorithms, and code trace (Xie et al., 2019). This review will focus on code tracing. As defined above, code tracing involves simulating the steps the computer takes to execute a program (Cunningham et al., 2017), including keeping track of variables and their values, user input, and program output. Figure 2.1 shows an example of a brief Python program (left) and the corresponding code trace (right). The code trace keeps track of variable values (e.g., see line 1), output (e.g., see line 2), input (e.g., see line 3) and how variables are updated (e.g., see line 4) when simulating program execution.

Code tracing can be challenging for students. Vainio and Sajariemi (2007) conducted weekly one-hour interviews with students enrolled in an introductory computer science course ($N = 6$). Four code-tracing difficulties emerged from the analysis of interview data: (1) single-value tracing, where students only used one value to update
all variables, instead of tracking each variable and its value; (2) confusing program function with its structure, where students made assumptions about the code based on its structures (e.g., expecting a counter’s value to be zero at the start of any given loop); (3) inability to abstract details of the code to identify its high-level function; (4) inability to use external representations of a program (e.g., via diagrams) to aid code tracing. These difficulties were attributed to a lack of experience and/or a need for further instruction on how programs work.

A working group on code-tracing instruction analyzed think-aloud transcripts of students ($N = 37$) solving multiple choice code-tracing questions (Fitzgerald et al., 2005). Students employed different strategies when responding to code-tracing questions. For example, some students used the pattern-recognition strategy to incorrectly infer a program’s output based on the output of similar programs. Other participants used a line-by-line code-trace strategy, but the code traces produced were often incomplete, describing only some lines of the code, or included errors. Some
students fail to code trace altogether. Lister et al. (2004) asked novice programming
students \((N = 556)\) to answer a code-tracing question and show their work. Only
63\% of students produced the correct response. Test responses from a subset of the
participants \((N = 37)\) were analyzed. Answers with explicit sketches of code traces
(e.g., sketches keeping track of variables and values throughout program execution)
were the most often correct, while answers with no sketches at all were the least often
correct.

Cunningham et al. (2019) interviewed 13 CS1 students following a test involving
five code-tracing questions to obtain insight into students’ code-tracing approaches.
The students were provided with the test questions and their scratch paper from the
test during the interview. Some students had followed the instructor’s code-tracing
style, while others used a different style because of a personal preference for another
style or to save time. Students indicated they sometimes had not used scratch paper
to code trace when solving the test’s code-tracing problems because they felt they
already understood the goal of the program, making the code trace unnecessary in
their opinion. Others produced incomplete code traces and said this was because they
stopped code tracing once they recognized the program’s pattern. This goal-oriented
approach is problematic since students were making assumptions about the computer’s
behaviour rather than monitoring each detail of program execution.

Cunningham et al. (2017) analyzed CS1 students’ \((N = 65)\) code-trace sketches
produced while solving five code-tracing problems and three code-generation problems.
Sketches that kept track of variables and their changing values resulted in correct
answers more often than other sketches. Making a code trace explicit through a sketch
may reduce cognitive load, thereby making the problem easier to manage and solve
without error. A complete lack of sketching resulted in the least success. Cunningham
et al. speculated that some of the issues students had with code tracing were due to inaccurate mental models of the notional machine, a construct we next describe.

### 2.2 Learning to Code Trace with Explicit and Implicit Notional Machine Instruction

A notional machine is an abstraction of how the computer executes a program (Du Boulay, 1986). Du Boulay (1986) noted that novices struggle to understand this concept and often expect the computer to execute a program based on what that program is intended to do (i.e., “intelligent mind” misconception, also described in Sorva, 2013) instead of understanding that the computer executes code exactly as written. Sorva (2013) proposes that a correct mental model of the notional machine is needed to accurately simulate program execution. A mental model is an internal representation that can simulate a dynamic process (Sorva, 2013). When code tracing, mental models of the notional machine are formed and refined whether these models are taught implicitly or explicitly. Sorva (2013) claims that altering an existing inaccurate mental model is more difficult than initially forming a correct mental model and so suggests that fostering accurate mental model formation is important. Du Boulay (1986) predicts that incorporating a notional machine into programming lessons can support mental model formation and help novice programmers understand how program execution occurs. Other work also advocates for direct instruction of the notional machine to support accurate mental model formation (Krishnamurthi and Fisler, 2019; Lowe, 2018; Robins, 2019; Sorva, 2013). While little work exists on how to explicitly teach the notional machine to novice programmers, there are several notable exceptions described below.
2.2.1 Designing Code-Tracing Lessons with Notional Machine Instruction

Mayer (1975) reported on three experiments testing instructional designs incorporating the notional machine. The first experiment compared learning outcomes of participants \( N = 80 \) learning to write and trace code with FORTRAN from different instructional materials: (1) model instruction, where the instruction used a notional machine model, (2) rule instruction, with definitions provided without a model, (3) rule-flow instruction, involving definitions and a flow chart, and (4) model-flow instruction, involving both a model and a flow chart. Participants were given instructional booklets that varied these materials, and then took a programming test involving code-tracing and code-generation questions. The two model groups performed better on code-tracing questions and on a looping code-generation question, while the other groups performed better on non-looping code generation questions. The second experiment \( N = 40 \) replicated these results and additionally examined the impact of pseudo mastery learning (where, following learning from the instructional booklet, participants had to generate correct solutions to a specified number of problems before moving on to subsequent exercises). The model group had relatively fewer code-tracing errors than the rule group, while the opposite was true for code generation. Experiment 3 \( N = 56 \) investigated the impact of instruction type (model versus rule), practice type (code generation or code tracing), and knowledge (based on a scholastic aptitude test). While there was little difference in learning between groups for high-knowledge students, for low-knowledge students, model instruction paired with code-generation practice was most beneficial, followed by rule instruction paired with code-tracing practice.
Bayman and Mayer (1988) compared learning of participants \( N = 95 \) in five groups, each given a different type of instruction: standard, transaction (detailed code-tracing instructions were provided in English), summary transaction (a summarized version of the transactional instructions), diagram (instruction included a notional machine diagram but inferences needed to be made about code execution details), and transaction diagram (with a notional machine diagram and the detailed code-tracing steps). The experiment was conducted over five one-hour sessions in a period of two weeks. The first session included a demographics questionnaire and description of the study, the second and third sessions involved participants learning BASIC programming from their designated instructional booklet, and the final two sessions tested participants’ programming knowledge, including code-tracing. For low ability students (based on math knowledge), the standard instruction resulted in the lowest scores, as compared to the other types of instruction. High-ability students performed similarly in all conditions. In this work, code-tracing ability was used as a proxy for mental model quality and so the authors argued that direct instruction about the notional machine results in better mental model formation.

Mayer (1989) surveyed 20 studies involving novices learning a variety of scientific topics (including computer programming and code tracing) from explanatory material and models. In the context of the review paper, a model was any diagram meant to encourage mental model formation. Based on a synthesis of the results, the inclusion of models increased conceptual information recall, with participants provided with a model scoring on average 57% better than participants not given models. The model group had lower verbatim recall scores than the group without a model but this difference was small (14%) and may have been the result of participants focusing on integrating the lesson content into their mental models, rather than focusing on the
lesson verbatim. The model group produced more creative solutions and solved more transfer questions, performing on average 64% better than non-model participants.

2.2.2 When to Integrate the Notional Machine in a Code-Tracing Lesson

Mayer (1976) investigated how the timing of notional machine presentation relative to other instructional text affects learning. The notional machine model included four components: memory scoreboard (an abstraction of computer memory), input window (an abstraction of the computer’s ability to accept user input), program list & pointer arrow (an abstraction of the computer executing lines of a program), and output pad (an abstraction of the computer’s output functionality). Mayer hypothesized that the notional machine would help students assimilate the lesson material in a meaningful way (i.e., form mental models of the notional machine). The first experiment manipulated the timing of the notional machine model presentation in instructional text. This was a $2 \times 2 \times 2$ design, with the first factor corresponding to whether the model was included before and during the lesson or not, the second factor corresponding to whether the model was available during testing or not, and the third factor corresponding to whether model or standard instruction was used. Participants ($N = 80$) learned programming from an instructional text that manipulated these factors and subsequently answered program-generation and code-tracing questions. When the model was presented prior to the instructional text, participants performed significantly better overall, supporting Mayer’s hypothesis. These groups performed particularly well on problems involving code tracing and loop generation. While results indicated no overall difference between learning outcomes for standard instruction and model instruction (when collapsing timing of model presentation), groups with model
instruction excelled on questions about looping while groups without model instruction
excelled on questions about individual functions (e.g., about the *print()* function but
not in the context of a larger program). In a second experiment, Mayer (1976)
replicated these results, showing that participants presented with the notional machine
model prior to the lesson performed significantly better overall than other groups. In
summary, these experiments demonstrate that there are benefits to providing students
with a notional machine model prior to the instructional lesson to serve as a meaningful
basis for assimilating the lesson concepts.

Mayer and Bromage (1980) conducted two experiments involving a notional machine
with the same key computer components as in Mayer (1976). In the first experiment,
participants \( N = 60 \) were presented with the notional machine prior to a code-tracing
lesson (before group) or following the lesson (after group). After the lesson, participants
were tested on their recall of lesson concepts. Mayer and Bromage hypothesized that
the notional machine components would act as conceptual anchors when presented
before the lesson. That is, they expected the notional machine components to act as a
basis to support learning and retention of the lesson. Participants in the before group
excelled at recall of concept units (facts from the text about locations in the computer
and processes it performs), production of novel summaries and appropriate reference
to the text and model, while the after group performed better on technical recall and
syntax details and produced more vague summaries and irrelevant references to the
text. Experiment 2 investigated if these effects persisted following a time delay. As in
experiment 1, participants \( N = 48 \) were presented with the model before or after
the lesson and were tested on recall of lesson concepts immediately after or following
a one-week delay. When comparing before and after groups, the result patterns
were similar to experiment 1. The before group produced many concept units, novel
summaries, and correct references (to relevant statements from the text). The results support Sorva (2013)’s proposal that it is more difficult to change an existing mental model than to form a new and correct mental model. In particular, students initially presented with the notional machine were able to integrate it into their understanding, while students presented with it after the lesson did not retroactively alter their mental models to reflect the notional machine components.

2.2.3 Research on Explicit Notional Machine Instruction

The aforementioned work provides evidence that instruction about the notional machine can help novices learn to code trace. However, there are relatively few studies and these date back over 30 years. Recently, the need for research on notional machine instruction has been emphasized, resulting in a working group on this topic (Fincher et al., 2020). The working group is in the process of assembling notional machine examples from various sources, including textbooks, classrooms, and interviews with teachers. The final report will present a catalogue of notional machines along with their descriptions.

2.2.4 Learning to Code Trace with Implicit Notional Machine Instruction

Work described above tested the impact of incorporating explicit representations of the notional machine into instruction. Other research investigates whether students can learn to code trace from instruction that does not explicitly incorporate a model of it. Instead, the instruction focuses on code tracing without an abstracted model of the computer. Thus, various computer components are not present, so students
must infer their existence. For example, if memory is not represented, a student needs to make assumptions about how the computer keeps track of variable values (e.g., they may accurately assume the computer keeps track of the most recent value or inaccurately believe that it keeps track of all previous values as well).

Hertz and Jump (2013) tested a new code-tracing-based teaching method, where each lesson topic included a code-trace example and half of the lecture time was devoted to students completing activities that always included at least some code-tracing practice. Key outcomes in these code-trace-based courses ($N = 9$ to 31, depending on the class) were compared to three courses from previous years taught in a standard way (with less focus on code tracing). The code-tracing method resulted in a significant decrease in dropout rates (from 25.49% in the previous years to 8.51% in the code-trace-based course), a slight increase in lecture grades, and a significant increase in lab grades.

Xie et al. (2018) tested whether teaching a code-tracing strategy would improve learning. The strategy reminded students to read each line of code while keeping track of variables and their values in a table and provided instructions on how to do so. The participants ($N = 24$) were university students taking an introductory programming course. The experimental group was taught the strategy by a human researcher using an instructional text while the control group was not. The strategy group outperformed the control group on a midterm test.
2.3 Computer Tools to Support Novice Programmers

2.3.1 Visual Debuggers

Visual debuggers, also known as animated debuggers or program visualizers, can guide code-tracing activities by visually representing the state of the program at each step of execution (i.e., performing a code trace) (Awasekar, 2013). While most debuggers aim to help expert programmers find errors in their code (Sorva, 2013), some debuggers have been developed to support novices learning to program (Yang et al., 2018; Kaila et al., 2010; Shi et al., 2017). To illustrate, a visual debugger system implemented by Heinsen Egan and McDonald (2014) was designed to support novice C program debugging activities through visualizations and explanations of bugs. Awasekar (2013) compared learning from a teacher using Learning Object (LO), a programming visualization tool to instruct C programming, versus learning from a standard chalkboard lesson. The LO group performed significantly better on a posttest than the group without LO. Perez-Schofield et al. (2019) conducted a study with C-Sim, a tool used to visualize program execution by graphically representing memory management for C programming. The results showed that students using C-Sim had higher exam performance than students without access to the tool. Smith and Webb (2000) developed a visualization tool called Bradman to support mental model formation related to C programming. While Bradman was appreciated by students, it did not universally improve learning outcomes (i.e., performance of the group with the tool was better on one task than the group without it but worse on a second task).
In a review of visual debuggers, Sorva et al. (2013) reported that these systems resulted in more learning than standard code tracing without such tools, but also reported some limitations of these tools. Notably, typical visual debuggers are designed to help experts effectively debug large programs and therefore do not focus on helping novice programmers understand the program. Sorva et al. propose that the cognitive load involved with learning how to use visual debuggers at the same time as learning to program may be challenging, especially for weaker students.

### 2.3.2 Intelligent Tutoring Systems

Another form of computer support is intelligent tutoring systems (ITS). ITS are computer tutors that provide personalized support to the user. This support can involve hints or feedback on responses. Most work to date in this area has focused on helping students write programs (Rivers and Koedinger, 2017; Price et al., 2017; Anderson et al., 1989; Bhuiyan et al., 1994; Fabic et al., 2019; Greer et al., 2001), for instance by generating hints (Rivers and Koedinger, 2017; Price et al., 2017), or providing feedback with explanations (Anderson et al., 1989). Less work exists on helping students code trace and none, to the best of our knowledge, on explicitly teaching the notional machine.

As far as ITS code-tracing support, Nelson et al. (2017) created PLTutor, a computer tutor that supports code tracing without explicit notional machine instruction. It allows students to step forward and backwards through computer programs using three types of assistance: (1) conceptual instruction, the tutor showing an explanation of the step; (2) execution, the tutor visually representing changes as a result of the step; and (3) assessment, the tutor prompting students to input values for the code trace. A two-condition study was used to evaluate the effectiveness of PLTutor by
comparing it to instruction provided by Codecademy, an online programming tutorial. Participants ($N = 37$) were undergraduate students beginning a CS1 course. There were two conditions, namely PLTutor and Codecademy. The PLTutor group had higher learning scores overall (but not significantly so) and significantly higher learning on specific questions (such as a code-tracing question involving conditionals and multiple variables) compared to Codecademy participants. This suggests that learning to code trace within the context of a tutoring system may be an effective way to support novice programmers.

Jennings and Muldner (2021b) implemented a tutor for novice programmers, called CT-tutor, to support code-tracing activities. CT-tutor included pairs of examples of code traces and similar corresponding problems. Four versions of the tutor were created to manipulate the level of interface scaffolding for the problems (high or reduced) and the instructional order (example-first or problem-first). The High-scaffolding interface supported code tracing by specifying all variables to be tracked, requiring variable values to be entered in a specific order, and giving feedback on intermediate steps. It also prompted users to explain the purpose of the program being traced. The reduced-scaffolding interface required participants to only enter the final values of variables (a free-form text box was provided that could be used to keep track of the code trace but this was not enforced); this interface did not include the explain prompt. An experiment ($N = 97$ participants) showed that CT-tutor increased learning overall, instructional order did not affect learning, and reduced scaffolding resulted in marginally more learning than high scaffolding. The latter result suggests that high scaffolding may have provided too much assistance and permitted shallow interaction with examples and problem solving.
2.4 Self-Explanation

A general learning strategy is self-explanation, the process of explaining instructional material to oneself by constructing inferences (Chi et al., 1994). Chi et al. (1989) investigated how students self-explained examples and solved problems in the physics domain \(N = 9\). A qualitative approach was used to identify self-explanations in participant utterances during example studying and problem solving. Based on problem-solving performance, the top four students were categorized as good students and the bottom four as poor students, with the middle student omitted. Good students generated self-explanations with inferences or with justification of example components. Poor students self-explained less, and their self-explanations were more superficial.

Given that not all students self-explain spontaneously, Chi et al. (1994) investigated if eliciting self-explanations would benefit learning. Eighth grade students \(N = 24\) took a pretest, read an expository biology text about the circulatory system, and then took a posttest. The study was conducted over three to four sessions, each of which lasted one to three hours. While they read the biology text, students were either (1) prompted to self-explain each line of the text without feedback or (2) unprompted, where they were asked to just read the text. The prompted group had significantly higher learning than the unprompted group, particularly on questions requiring inference. Chi et al. (1994) categorized students within the prompted group as high or low explainers based on the number of self-explanations produced. All high explainers had the most accurate mental models of the circulatory system, while only one of four of the low explainers did.

Schworm and Renkl (2006) investigated learning of participants \(N = 80\) using a \(2 \times 2\) factorial design, with the first factor being learning with or without self-explanation prompts and the second factor being learning with or without instructional
explanations. Prompting for self-explanations resulted in more learning than providing instructional explanations. Self-explanation prompts were helpful despite students not receiving feedback.

In a meta-analysis, Bisra et al. (2018) reviewed research that experimentally compared learning with self-explanation prompts to learning without self-explanation prompts. The meta-analysis confirmed that self-explanation prompts increase learning for both declarative and procedural domains. Rittle-Johnson et al. (2017)’s meta-analysis in the domain of mathematics education also showed that prompting for self-explanation is beneficial to learning. The benefits of prompting for self-explanation increased with high-quality scaffolded explanations. Scaffolding can be achieved via self-explanation training or by structuring explanations to direct student attention (e.g., by providing a dropdown list of possible self-explanations or by including more specific prompts).

2.4.1 Self-Explanation and Code Tracing

Several studies have investigated self-explanation in the context of code-tracing activities (Jennings and Muldner, 2021a; Zhi et al., 2019; Kumar, 2014). Kumar (2014) tested the effect of prompting for self-explanation during code-tracing activities in a tutoring system. Students \((N = 730)\) used the tutor on their own time and unsupervised. When a question was answered incorrectly, the experimental group was provided with feedback and a step-by-step worked code-tracing example including self-explanation prompts. The control group received feedback and a code-tracing example but no self-explanation prompts. Learning was similar between groups, perhaps because the control group engaged in unprompted self-explanation when studying the example.
Vihavainen et al. (2015) compared learning of students \((N = 118)\) in an introductory programming course using three conditions: learning without self-explanation prompts, learning with self-explanation prompts, and learning with self-explanation multiple-choice prompts (i.e., questions where self-explanation is required and the final answer is selected from a set of options). The questions involved code-tracing programs to predict their output. Both self-explanation groups performed significantly better than the no-self-explanation group on exam explanation questions.

Chung and Hsiao (2021) analyzed students’ \((N = 112)\) responses to multiple-choice questions, including code-tracing questions, in an online platform. Students were presented with one question per day. After responding to the question, students could self-explain how they obtained their answer, but this was optional. Students were labelled as low or high performing based on exam grades. When a question in the online platform had been self-explained, low performing students’ answers had a significantly lower error rate than when they had not been self-explained. So, low-performing students benefitted from self-explaining while code tracing.

### 2.5 Summary

Code tracing is an important skill yet novice programmers struggle to code trace. This difficulty may be due to novices forming inaccurate mental models of the notional machine. Introducing an explicit notional machine at the beginning of a code-tracing lesson can help students form an accurate mental model of the notional machine, however little work exists in this area (with some exceptions dating back over 30 years). This thesis aims to investigate the utility of explicit notional machine instruction for novice programmers learning to code trace, within the context of a tutoring system.
with self-explanation prompts.
Chapter 3

Pilot

Before running the main study, we conducted a pilot to inform instructional material and notional machine design for the lesson used in the final study (Chiarelli and Muldner, 2021, 2022a).

3.1 Participants

Participants \((N = 8)\) were recruited via word-of-mouth by the thesis author. All participants were novice programmers and were 18 years of age or older. Participants were compensated $25 for their time.

3.2 Materials

3.2.1 Python Programming Lesson

A lesson was created to teach about several fundamental programming topics, namely variables, input and output, and while loops. The lesson was a 12-page document
that described these concepts (see Appendix A.3). Five self-explanation prompts were distributed throughout the lesson to encourage participants to solidify the material presented and to provide insight into their mental models (see Table 3.1).

Table 3.1: Self-Explanation Prompts Used in the Lesson.

<table>
<thead>
<tr>
<th>Prompt Number</th>
<th>Context</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>After a brief introduction and a description of the notional machine and its components</td>
<td>At this point, how do you think the computer runs a program?</td>
</tr>
<tr>
<td>2</td>
<td>1 city = &quot;ottawa&quot;</td>
<td>What happens when this program is “run”?</td>
</tr>
<tr>
<td></td>
<td>1 counter = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 password = input(&quot;google password &quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 counter = counter + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 password = input(&quot;amazon password &quot;);</td>
<td>What do you think the model would look like for the steps of this program?</td>
</tr>
<tr>
<td></td>
<td>5 counter = counter + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 print(counter)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 print(password)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 response = &quot;--&quot;</td>
<td>Can you explain in your own words what happens when this program is “run”?</td>
</tr>
<tr>
<td></td>
<td>2 while response != &quot;test123&quot;:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 response = input(&quot;Enter password: &quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 print(&quot;hmm&quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 print(&quot;Welcome&quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>At the end of the lesson</td>
<td>Now, how do you think the computer runs a program?</td>
</tr>
</tbody>
</table>

The notional machine was used in the lesson to illustrate, at a high level, how the computer executes a program. Its design was based on Mayer (1976)’s notional machine and was depicted as a large rectangle containing four components, namely
memory, input, output, and program pointer (see Figure 3.1). Memory was represented as a series of four boxes along the top of the notional machine. These boxes stored a program’s variables and their values – the lesson described how this “memory” was updated when program variable values changed. Input was represented as a rectangle (see bottom left, Figure 3.1 and Figure 3.2). When a program asked for user input, the user’s input was added to the input box. Output was represented as a box in the notional machine (see bottom right, Figure 3.1 and Figure 3.2). The program to be code traced was shown in a rectangle in the middle of the notional machine, along with numbered program lines and a program pointer indicating the current line being traced (see Figure 3.2).

The programming lesson was text based and shared as a PDF document. It began with a brief introduction to programming and then presented the notional machine and its components, including an explanation of how to use the notional machine to simulate program execution. The first self-explanation prompt asked participants to describe how the computer runs a program (see prompt 1, Table 3.1). Next, variables were described, including how to create and update them in Python, illustrated with brief Python programs. Each program was accompanied with a code trace of the
Figure 3.2: State of the Notional Machine After Executing The First Four Lines of a Seven-Line Program Involving a While Loop.

program. Some code traces were presented within the context of the notional machine, and so included the notional machine’s state following the execution of each line of code (i.e., a one-line program was code traced with one notional machine state; a two-line program with two notional machine states and so on). Other program code traces did not include explicit reference to the notional machine to keep the lesson document a reasonable length.

The variable section of the lesson included one prompt asking what happens when a sample one-line program is run (see prompt 2, Table 3.1). The remaining two sections of the lesson covered user input, output, and loops, and followed the same format: concepts were described and brief programs were code traced, some within the context of the notional machine. Each section included a self-explanation prompt to solidify concepts presented. The lesson finished with a prompt asking participants to describe how the computer runs a program (see prompt 5, Table 3.1).
3.2.2 Pretest and Posttest

A pretest (see Appendix A.2) and posttest (see Appendix A.4) were created to measure learning. Each test contained eight programming questions covering the three topics of the lesson (variables, input and output, and while loops). The two tests were isomorphic, differing only in superficial aspects corresponding to variable names, strings, or values.

3.2.3 Interview

A five-question semi-structured interview was developed to obtain participant feedback about the notional machine and general lesson design (see Table 3.2 for questions).

Table 3.2: Interview Questions.

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>What did you like?</td>
</tr>
<tr>
<td>What did you not like?</td>
</tr>
<tr>
<td>Was the model useful or distracting?</td>
</tr>
<tr>
<td>Is there anything you would change about the model</td>
</tr>
<tr>
<td>Do you have any other feedback?</td>
</tr>
</tbody>
</table>

3.3 Procedure

The pilot was conducted one-on-one via Zoom. Each session lasted up to two hours. Participants first signed the informed consent form. They were then introduced to the experiment and given an overview of the process. Next, participants confirmed that they had not completed any programming courses and they were given approximately 15 minutes to complete the pretest.
Following the pretest, the researcher screenshared the lesson and participants were given control of the screen in order to scroll through the lesson at their own pace as they studied it. We used the think-aloud protocol: Participants were asked to read aloud, to say whatever was on their mind as they studied the lesson, and to respond to self-explanation prompts orally. Throughout the lesson, if participants were quiet for more than approximately five seconds the experimenter prompted them to self-explain with general prompts such as: “What are you thinking of now?”; “What are you reading now?”; or “Please keep talking”. This phase lasted about 75 minutes, with optional short breaks offered following the first two lesson sections.

After completing the lesson, participants were asked to answer the exit interview questions and provide feedback on the lesson materials. The exit interview lasted approximately 10 minutes. Both the lesson and interview were audio recorded.

As the final step, participants completed the posttest and were given 20 minutes to do so.

3.3.1 Data Preparation and Coding Scheme

We had two high level goals: (1) to gather information on how to improve the instructional material for the subsequent version of the lesson and (2) to identify how prompting for self-explanation affected the quantity, quality, and content of self-explanations produced. All participant utterances were transcribed and then analyzed. We had the following research questions:

1. Does prompting during the lesson encourage engagement/self-explaining?

2. Does prompting result in better quality self-explanations (e.g., more complete explanations of concepts)?
3. Do participants refer to the notional machine more frequently when prompted to self-explain?

4. Do participants learn from the lesson?

5. What feedback do participants have about the lesson?

To address these questions, we used a qualitative approach. A coding scheme was developed to categorize utterances. As in Chi (2000), domain specific utterances that went beyond the instructional text were labelled as self-explanations. This included paraphrases and inferences. The coding-scheme further characterized self-explanations as follows: (1) spontaneous self-explanation with notional machine intrusion, (2) spontaneous self-explanation without notional machine intrusion, (3) prompted self-explanation with notional machine intrusion, and (4) prompted self-explanation without notional machine intrusion. Prompted self-explanations were provided in response to the lesson’s prompts, while spontaneous ones occurred without prompts. Any mention of the notional machine or its components during self-explanation was labelled as a notional machine intrusion, which indicated that participants may be integrating notional machine components into their mental models. Table 3.3 presents examples of each type of self-explanation. The coding scheme was used by the thesis author to identify and categorize self-explanations in the transcripts.
Table 3.3: Self-Explanation Categories and Sample Participant Self-Explanations, with Notional Machine Intrusions in **Bold**

<table>
<thead>
<tr>
<th>Self-Explanation Category</th>
<th>Topic</th>
<th>Prompt</th>
<th>Sample Self-Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spontaneous self-explanation without notional machine intrusion</td>
<td>While loops</td>
<td>N/A</td>
<td>P7: “If you needed, I don’t know, a postal code or if you needed a country, and it wasn’t allowed in that particular one then it’s going to wait until you put the right thing in there.”</td>
</tr>
<tr>
<td>Spontaneous self-explanation with notional machine intrusion</td>
<td>Input</td>
<td>N/A</td>
<td>P5: “Okay, so whatever I put in the <strong>input</strong> would go straight to the <strong>memory</strong>”</td>
</tr>
<tr>
<td>Prompted self-explanation without notional machine intrusion</td>
<td>While loops</td>
<td>Can you explain in your own words what happens when this program is “run”? (see 3.1 context for prompt 4)</td>
<td>P3: “It’s when I keep messing up my password, so it keeps asking me to enter it in. So, it keeps saying “is the condition true? Yes,” because it’s not right. So, it loops me back to the beginning. And then once I get it right, I don’t need to continue doing that loop. So, it just brings me right to step five.”</td>
</tr>
<tr>
<td>Prompted self-explanation with notional machine intrusion</td>
<td>General</td>
<td>How do you think the computer runs a program? (see Table 3.1 context for prompt 1)</td>
<td>P2: “Okay, so I think it’s like our brain. We see something, that’s <strong>input</strong>. And then we try to memorize what we saw. And then at one point, if we need to recall it, we have to go back to our <strong>memory</strong>, but we have to do it in a linear way. So, it will once the <strong>output</strong> comes it. It has to be very clear when we program so it has to be ABC. We can’t go ACB.”</td>
</tr>
</tbody>
</table>
3.4 Results

Table 3.4 shows the total number of each type of self-explanation per participant as well as the overall mean and standard deviation across participants. Each participant produced more spontaneous self-explanations than prompted ones (see the mean column in Table 3.4), which is unsurprising since the lesson afforded a maximum of five prompted self-explanations (in response to the five prompts), while any part of the 12-page lesson could have been spontaneously self-explained.

Table 3.4: Self-Explanations in Each Category, per Participant

<table>
<thead>
<tr>
<th>Self-Explanation Category</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>M</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spontaneous self-explanation without notional machine intrusion</td>
<td>4</td>
<td>44</td>
<td>18</td>
<td>62</td>
<td>65</td>
<td>66</td>
<td>68</td>
<td>30</td>
<td>44.63</td>
<td>24.77</td>
</tr>
<tr>
<td>Spontaneous self-explanation with notional machine intrusion</td>
<td>3</td>
<td>9</td>
<td>21</td>
<td>25</td>
<td>20</td>
<td>29</td>
<td>24</td>
<td>6</td>
<td>17.13</td>
<td>9.73</td>
</tr>
<tr>
<td>Prompted self-explanation without notional machine intrusion</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1.63</td>
<td>0.92</td>
</tr>
<tr>
<td>Prompted self-explanation with notional machine intrusion</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2.38</td>
<td>1.06</td>
</tr>
</tbody>
</table>
3.4.1 Does Prompting During the Lesson Encourage Engagement/Self-Explaining?

The results suggest that the prompts encouraged self-explanation for participants who did not spontaneously self-explain much. For example, P1 did not spontaneously self-explain frequently with only seven spontaneous self-explanations throughout the lesson, yet they self-explained in response to four of the five self-explanation prompts (see column P1 in Table 3.4).

In general, all participants answered more than half of the prompted self-explanations, but only two answered all five (four answered only four and two answered three). Thus, prompting often but not always led to self-explanation for one of two reasons (1) the participant did not know the answer or was overly confused (e.g., P3 said “I have no idea what happens when example three is run” in response to a prompt asking to explain how a program runs); or (2) the participant already self-explained the information targeted by the prompt (e.g., P4 had already code traced a program and then was prompted to do so. P4 said “Well, it’s been answered.”). As the latter point illustrates, prompting did not lead to redundant explanations for participants who had already spontaneously self-explained the concepts since they just moved along in those cases.

3.4.2 Does Prompting Result in Better Quality Self-Explanationss (e.g., More Complete Explanations of Concepts)?

In some cases, prompting produced complete self-explanations, which described entire concepts rather than fragments of a concept or program. For example, a spontaneous self-explanation explained the input function only, while a prompted self-explanation
explained how several computer components from the lesson interact to run a program (see Spontaneous self-explanation with notional machine intrusion versus Prompted self-explanation with notional machine intrusion in Table 3.3).

3.4.3 Do Participants Refer to the Notional Machine More Frequently When Prompted to Self-Explain?

All participants produced self-explanations with notional machine intrusions (see Table 3.4), so some aspects of the notional machine were incorporated into their understanding of how the computer runs a program. Table 3.4 shows that, on average, a larger proportion of prompted self-explanations involved notional machine intrusions compared to spontaneous ones. This supports the inclusion of prompts, suggesting that participants make more explicit connections between programs and notional machine components when prompted to self-explain. Some self-explanations included notional machine intrusions even when the lesson did not refer to the notional machine for those examples. For example, when prompted to explain what happens when `city = "ottawa"` is run, P6 said “The word city, Ottawa will be stored in the memory model above”, referring to the notional machine memory component even though the notional machine was not included in that example in the lesson. However, not all self-explanations (prompted or spontaneous) included notional machine intrusions. It is unclear if that is because the notional machine components were not fully incorporated into student mental models of the notional machine or simply because it takes more time to refer to components of the notional machine while code tracing (e.g., when describing `counter = 0`, it is faster to say `the value of counter is 0` than to say `counter and its value of 0 are stored in the computer’s memory`).
3.4.4 Did Participants Learn From the Lesson?

The pretest and posttest were scored so that each question was worth one point if answered correctly, with no part marks given. Table 3.5 shows the pretest and posttest scores and learning (the increase in scores from pretest to posttest) for each participant as well as the mean and standard deviation of the scores for each category. Each participant learned (their score improved from pretest to posttest), but all scores were low, with the highest participant score at 4/8. Thus, difficult concepts need to be better described in the lesson. Specifically, no participant correctly answered any of the four pretest or posttest questions about predicting the output of while loops. So, the scores in Table 3.5 reflect understanding of variables, input, and output (the first four of eight posttest questions) since all participants scored zero on all of the last four posttest questions that were about while loops. Based on analysis of responses, none of the participants returned to the beginning of the while loop to check the loop condition. To illustrate, Figure 3.3 shows a while loop program that participants were asked to code trace on the posttest. A correct code trace would indicate that line 1 was executed once followed by lines 2, 3 and 4, that were all repeated three times (because the condition on line 2 is true until the loop has iterated three times, incrementing the value of total to 3); once the loop terminated, program lines 5 and 6 were executed once. However, participants’ code traces indicated that lines 1 thorough 8 were sequentially executed once, without any repetition or reference to the condition on line 2 following execution of line 4. This lack of understanding of repetition in while loops was surprising because self-explanations throughout the exercise showed some understanding of the repetitive nature of the loop (see Prompted self-explanation without notional machine intrusion in Table 3.3 for an example). To address this issue, we revised the lesson for the final study to emphasize repetition in the context
Table 3.5: Number of Correct Answers (Out of 8 Questions) on Pretest and Posttest and Learning (Increase in Scores from Pretest to Posttest), per Participant

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>M</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pretest</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.63</td>
<td>0.92</td>
</tr>
<tr>
<td>Posttest</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2.75</td>
<td>0.71</td>
</tr>
<tr>
<td>Learning</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2.13</td>
<td>1.13</td>
</tr>
</tbody>
</table>

of a loop and included self-explanation prompts directing student attention to this concept.

Other issues included participants confusing their expectations of how a program was intended to work with how the program was actually written. For example, print statements like `print("you entered:")` confused some participants, who assumed that line of code would print what the user had entered rather than the string containing the words “you entered:”.

```plaintext
1 total = 0
2 while total < 3:
3     total = total + 1
4     print(total)
5 print("ha")
6 print(total)
```

Figure 3.3: A While Loop Program to Code Trace on the Posttest.

### 3.4.5 What Feedback Do Participants Have About the Lesson?

In the interview, participants reported liking the notional machine “I don’t know that I would have understood what was going on without the pictures”. One participant even reported having drawn the notional machine while code tracing during the posttest
“to do the questions I actually drew it out. Having the visual was helpful.” Another participant said “This was good because it shows me what’s going on in the computer.” Thus, they may have used the notional machine model as a conceptual anchor that provided an abstracted explanation of how the computer works, and not just as a tool to superficially organize the information in the program.

Participants had suggestions for improving the design and presentation of the notional machine. Some reported that at the beginning of the lesson they had difficulty determining which component belonged to which label in the notional machine “at first I thought input was this first box . . . and I thought output was this up here”. Some participants also asked that it be clearer where changes had been made to the notional machine from one state to the next “like just something that to highlight the fact that this is where things can get caught up.” Moving from text-based to another medium like a video may help with this issue.
Chapter 4

Methods

We now describe the main study we conducted to evaluate the notional machine lesson, starting with the methods.

4.1 Participants

The study participants were 48 individuals (16 males, 31 females, 1 agender) 18 years of age or older. Participants’ educational backgrounds were varied, with 11 participants from Arts and Social Sciences, four from Engineering and Design, eight from Public Affairs, 11 from Science, eight from Business, and six reported other/not applicable backgrounds. We recruited 16 participants via SONA (Carleton University’s experiment management system), 28 participants via Facebook posts on two Carleton University Facebook pages, and four participants via email (used for friends and family only). SONA participants were compensated with 2% credit in an introductory cognitive science course while other participants were compensated $25. In order to be eligible for the study, participants needed to be novice programmers, which we operationalized as having taken one university level computer science course or less or
having no prior programming experience.

4.2 Materials

4.2.1 Questionnaires: Demographics, Pretest and Posttest

A questionnaire was created using Google Forms to collect basic demographic information about participants (see Appendix B.2).

A pretest (see Appendix B.3) and posttest (see Appendix B.5) were created using Google Forms to measure learning of target concepts. The pretest and posttest each included seven questions targeting the topics covered in the programming lesson (variables, user input, output and while loops). The pretest and posttest questions were isomorphic but varied in superficial features, such as variable names and constants assigned to them. The questions measured code-tracing knowledge based on correctness of answers. Success on code-tracing questions was also a proxy for the accuracy of participants’ mental models of the notional machine (as in Bayman and Mayer, 1988). The seven questions each presented a Python program. Six questions asked participants to show what happens when the computer executes the program and to justify their answer by referring to each line of code. Questions 1 – 4 were basic code-tracing questions without loops (e.g., programs with variables being assigned and reassigned values). Question 4 included unexpected behaviour: a variable that incorrectly counted the number of input statements (see Figure 4.1 for the program to be code traced in pretest question 4). This was done to determine if participants accurately code traced (and predicted that line 9 would print 3) or if they incorrectly predicted that line 9 would print 2. The latter can be due to the misconception that the computer “understands” the purpose of a given program beyond the program
instructions (Du Boulay, 1986; Sorva, 2013). Question 5 was a multiple-choice question asking how many times a line inside of a while loop was printed. Question 6 was a basic code-tracing question involving a while loop. Question 7 was a more complex code-tracing question that also included a counter outside of the while loop (the example programs in the lesson only included a counter within the while loop). This was done to test indentation understanding (i.e., which lines of code are inside vs. outside a loop).

4.2.2 Programming Lesson

We developed a programming lesson consisting of instructional videos and self-explanation prompts on target concepts. The lesson was delivered through a tutoring system we implemented. Two versions of the lesson were created: a control version that did not include the notional machine and an experimental version that did include the notional machine. We begin with a description of the notional machine used in the experimental version of the lesson – this is the finalized version that was refined after the pilot.
4.2.3 Notional Machine

The notional machine design was informed by Mayer (1976)’s notional machine model and by participant feedback from the pilot study (see Figure 4.2). As in the pilot study, the notional machine included abstractions of four main computer components. To recap, the program pointer is an arrow pointing to the line of code currently being executed in the program shown in this component (see blue component, Figure 4.2, bottom centre). The memory area has four memory slots holding program variables and their values (see green component, Figure 4.2, top). Only four memory slots were included because this provides sufficient space for the programs in the lesson. The input box stores information provided to the program by the user (see orange rectangle, Figure 4.2, bottom left). The output box displays information produced by the program (see red 3D box, Figure 4.2, bottom right).

The notional machine was used in the lesson to demonstrate at a high level the process the computer takes when executing a program, thus illustrating what is needed to code trace the program. To make this concrete, we will illustrate with an example...

Figure 4.2: Notional Machine Used in the Experimental Group’s Programming Lesson.
for the basic two-line program shown in Figure 4.3. In the lesson, each program line produces a new, updated notional machine state. Since this program has two lines, there are two notional machine states. Figure 4.4 shows the state of the notional machine after the first line was executed: the program pointer indicates line 1 and the variable person appears in the first memory slot along with its value “mary”. After the second line of the program was executed, the program pointer moved to line 2 (see Figure 4.5) and the notional machine was updated by adding the string mary to the output component (see Figure 4.5, bottom right). Based on feedback from the pilot study, the most recent change made in the notional machine is highlighted by outlining the revised information in yellow (see Figure 4.4 and Figure 4.5).

```
1 person = "mary"
2 print(person)
```

Figure 4.3: A Two-Line Python Program.

Figure 4.4: State of the Notional Machine Following the Execution of Line 1 in the Program, with Changes Resulting from Line 1 Highlighted in Yellow.
4.2.4 Instructional Videos

Six brief instructional videos were created to teach basic programming concepts using the language Python. Each video showed content presented using a PowerPoint presentation accompanied by a small window displaying the “instructor” (see Figure 4.6 for an example). Video 1 introduced the concept of variables, explained variable assignment, and showed examples of simple programs involving only assignment. Video 2 demonstrated how to modify values of variables. Video 3 shifted the focus to user input and output, including how to get input from the user and how to show program output on the screen. Video 4 introduced while loops. The while loop repeated instructions based on the condition that followed the `while` keyword but did not include break statements or conditional statements (e.g., see program in Figure 4.7). Video 5 introduced conditional statements, including Boolean expressions in Python. Video 6 returned to the concept of while loops, extending the example program from Video 4 to include a counter variable that was initialized with a value of
zero and then increased by one at each iteration of the while loop. All videos included demonstrations of how to code trace the programs in the videos.

We created two versions of the six videos: one with a notional machine (experimental version) and one without (control version). The same content was included in both versions, but only the experimental videos used the notional machine during code tracing. The control videos illustrated code tracing in a more traditional way, that is, monitoring program variables and values using a code-trace table, and tracking output while simulating program execution line-by-line (see Figure 4.8). The experimental video lessons (38mins 27secs total) were longer than the control video lessons (27mins 40secs total) (see Table 4.1 for breakdown of individual video lengths).
Figure 4.8: An Example of a Control Video Demonstrating Standard Code Tracing with a Code-Trace Table (Orange Table on the Right Side of the Screen).

Table 4.1: Instructional Video Lengths for Experimental and Control Versions.

<table>
<thead>
<tr>
<th>Video</th>
<th>Experimental Version: Length (mins)</th>
<th>Control Version: Length (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video 1: Variables</td>
<td>8:25</td>
<td>5:00</td>
</tr>
<tr>
<td>Video 2: Variables Part 2</td>
<td>5:42</td>
<td>4:05</td>
</tr>
<tr>
<td>Video 3: Input/Output</td>
<td>7:02</td>
<td>4:31</td>
</tr>
<tr>
<td>Video 4: While Loops</td>
<td>7:56</td>
<td>6:17</td>
</tr>
<tr>
<td>Video 5: Conditionals</td>
<td>3:24</td>
<td>3:09</td>
</tr>
<tr>
<td>Video 6: While Loops Part 2</td>
<td>5:58</td>
<td>4:38</td>
</tr>
</tbody>
</table>
4.2.5 Self-Explanation Prompts

Nine self-explanation prompt screens were developed to encourage participants to actively engage with the instructional videos. Self-explanation prompts were identical for both lesson versions. The prompts were chosen based on existing research and based on participant feedback from our pilot study. Since prior research shows specific prompts result in more learning than general prompts (Bisra et al., 2018), the self-explanation prompts request explanation of specific concepts (e.g., What makes a while loop stop repeating?) or (parts of) programs (e.g., referring to Figure 4.9: What happens after line 5 is executed?) - see Appendix B.4 for the full list of the prompts. Further, the prompts were designed to direct attention towards concepts that are difficult for novice programmers. For example, students in the pilot study did not grasp the repetitive nature of while loops so we included prompts to reinforce this concept.

4.2.6 CTAT Tutor

The videos and the self-explanation prompts comprised the programming lesson. To present this lesson, we implemented a Cognitive Tutor Authoring Tools (CTAT) tutor. The advantages of using a tutor include the ability to specify the tutor behavior (e.g., indicating acceptable student responses) and automatic logging of student actions

```python
1   user = "--"
2   counter = 0
3   while user != "admin":
4       user = input("enter user type:"
5       counter = counter + 1
6   print("Welcome")
7   print(counter)
```

Figure 4.9: A Program with a While Loop, Used in a Prompt.
The tutor consisted of 20 screens making up the lesson. Each screen showed the current screen number and the total number of screens to help participants monitor their progress. Participants advanced through the lesson by clicking the *Done* button on the bottom of each page (see Figure 4.10). Participants were not able to return to previous screens once they moved on to a new one. To remind them of this, the tutor presented a message after the *Done* button was clicked asking the user to confirm that they finished with the current activity (see Figure 4.11).

Three types of screens were included in the tutor: instruction screens, video screens, and prompt screens. Instruction screens informed participants about logistics related to the lesson (see Figure 4.10 for an example). The tutor included five instruction screens (a welcome screen, a screen for participants to enter their ID, two screens indicating that participants may take a short break, and a screen signalling the end of...
Figure 4.11: A Message from the Tutor Confirming that the User is Done After They Have Clicked the Done Button.

The tutor included six video screens. Each video screen contained instructions asking participants to watch the video in its entirety, the video itself, and a done button (see Figure 4.12 for an example). The videos auto-played when the screen first loaded and participants had the option of making the video full screen.

The tutor also included nine prompt screens that followed a video lesson. These screens involved one or more prompts accompanied by text boxes used to provide responses (see Figure 4.13). Some prompts asked for explanations about each line of a multi-line program – in this case, a separate text box was included next to each program line (see Figure 4.14). At the bottom of the prompt screens, the most recent lesson video was included to provide participants with the option of reviewing the video lesson. For the prompt screens, participants could only progress to the next
Watch the video below. Please don’t skip ahead. Only press the done button when you have watched the entire video.

**PROGRAMMING BASICS: OUTPUT**

- Print: shows the output of a program to the screen
- Example 8: A simple program using print:

```python
1 print("hello")
```

- What happens when we run the program in Example 8?

---

Figure 4.12: A Video Screen in the Tutor.
Figure 4.13: A Prompt Screen in the Tutor.

screen if they had entered a response into each of the text boxes provided. If a prompt was skipped but a participant tried to leave by clicking the done button, the tutor displayed a message asking them to keep working. The tutor did not check if the responses were correct and so any input was sufficient to progress to the next screen.

The three types of screens were sequenced so that video screens were followed by corresponding prompt screen(s) and instructional screens were included as necessary (see Table 4.2).
Figure 4.14: A Prompt Screen in the Tutor with Response Boxes Next to Each Line of the Program Being Code Traced.
<table>
<thead>
<tr>
<th>Screen Number</th>
<th>Screen Type</th>
<th>Screen Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Instruction screen</td>
<td>Introduction and instructions</td>
</tr>
<tr>
<td>2</td>
<td>Instruction screen</td>
<td>Enter participant ID</td>
</tr>
<tr>
<td>3</td>
<td>Video screen</td>
<td>Video 1</td>
</tr>
<tr>
<td>4</td>
<td>Prompt screen</td>
<td>Prompt 1</td>
</tr>
<tr>
<td>5</td>
<td>Video screen</td>
<td>Video 2</td>
</tr>
<tr>
<td>6</td>
<td>Prompt screen</td>
<td>Prompt 2</td>
</tr>
<tr>
<td>7</td>
<td>Instruction screen</td>
<td>Three-minute break</td>
</tr>
<tr>
<td>8</td>
<td>Video screen</td>
<td>Video 3</td>
</tr>
<tr>
<td>9</td>
<td>Prompt screen</td>
<td>Prompt 3</td>
</tr>
<tr>
<td>10</td>
<td>Video screen</td>
<td>Video 4</td>
</tr>
<tr>
<td>11</td>
<td>Prompt screen</td>
<td>Prompt 4</td>
</tr>
<tr>
<td>12</td>
<td>Prompt screen</td>
<td>Prompt 5</td>
</tr>
<tr>
<td>13</td>
<td>Prompt screen</td>
<td>Prompt 6</td>
</tr>
<tr>
<td>14</td>
<td>Instruction screen</td>
<td>Three-minute break</td>
</tr>
<tr>
<td>15</td>
<td>Video screen</td>
<td>Video 5</td>
</tr>
<tr>
<td>16</td>
<td>Prompt screen</td>
<td>Prompt 7</td>
</tr>
<tr>
<td>17</td>
<td>Video screen</td>
<td>Video 6</td>
</tr>
<tr>
<td>18</td>
<td>Prompt screen</td>
<td>Prompt 8</td>
</tr>
<tr>
<td>19</td>
<td>Prompt screen</td>
<td>Prompt 9</td>
</tr>
<tr>
<td>20</td>
<td>Instruction screen</td>
<td>End screen</td>
</tr>
</tbody>
</table>

Table 4.2: Sequence of Tutor Screens.
4.2.7 CTAT Tutor: Implementation Details

To implement the tutor we created the tutor interfaces and behavior graphs that determined tutor responses (details below). A unique interface was created for each tutor screen using CTAT HTML Editor, a drag-and-drop application that allows the user to insert widgets for each interface without the need for programming. Once the file was saved, the editor automatically created the corresponding HTML and CSS files. To define how the tutor responds to student actions in the tutor, a behaviour graph was created for each tutor interface. A behavior graph is a representation of the valid ways to interact with a tutor interface (Aleven et al., 2009). For each interface, we used the CTAT application to demonstrate the steps the user needed to take to complete that interface’s activities and the tool created the behaviour graph based on those actions. Behaviour graphs were edited as necessary once they were created (e.g., accepting any student responses to prompts as correct; accepting responses in any order).

Figure 4.15 shows an example of an HTML interface and its corresponding behaviour graph. The behaviour graph encodes that the user will type into the first and second textbox (see (1) and (2) on interface and behaviour graph), and click the Done button (see (3) on interface and behaviour graph). Even though these steps are ordered in the graph, the unordered option was checked (see (4) in Figure 4.15), meaning that student responses are accepted in any order. The reenterable option is also checked (see (5) in Figure 4.15), meaning that students can edit their answer before pressing Done. The graph was refined (see asterisk next to (1) and (2) in Figure 4.15) so that any input is acceptable.

Two versions of the tutor were implemented. The versions differed only in the video lessons attached to the screens (i.e., experimental version included video lessons
with the notional machine and control version included video lessons with a code-trace table).

Both tutor versions were deployed to TutorShop, meaning they could be launched in a browser and so accessible over the internet. TutorShop “student” accounts were created and assigned either to the experimental tutor or the control tutor. All CTAT tutor behaviour (e.g., button clicks, response time) and response data (i.e., responses to prompts) from participants was automatically logged through DataShop.

### 4.3 Study Design and Procedure

A between-subject design was used with a control group and an experimental group. Participants were each assigned to a condition using a round-robin approach. The experiment was conducted in one-on-one Zoom video conferencing sessions lasting up to two hours per participant. All sessions were conducted by the thesis author.

To begin, participants were asked to read the consent form, given the opportunity
to ask questions, and, if they consented, to sign the informed consent form. They were then given a brief overview of the study and assigned a participant ID. Participants completed the demographics questionnaire and the pretest (15 minutes). Following test completion, the main phase of the experiment began. To reduce the feeling of being observed, participants were invited to turn off their camera and the experimenter also turned off their camera. Participants were then sent the link to a CTAT tutor (either the control tutor or the experimental tutor, depending on the assigned condition). They were asked to share their screen so that the experimenter could intervene if any issues arose (the experimenter’s camera was not on so this set up is similar to a lab study where the experimenter can see participants’ screens). Participants were instructed to work through the lesson in the tutor by watching the videos and answering the self-explanation prompts. They were told to spend approximately three minutes answering each self-explanation prompt screen but this was not enforced. Participants were not permitted to ask questions regarding the lesson content to ensure all participants received the same information. The lesson phase lasted approximately an hour.

After completing the lesson, participants were sent a link to the posttest. They were asked to answer all of the questions and told that following test submission they would be asked to orally explain their answer to the final posttest question. Participants had up to 20 minutes to complete the posttest. As the final step, participants explained their answer to the final posttest question – this phase was audio recorded.
Chapter 5

Results

The analysis was driven by these research questions:

1. Does explicit instruction about the notional machine result in more learning than instruction without the notional machine?

2. Does explicit instruction about the notional machine when learning to code trace result in more learning on easy questions and on hard questions than instruction without the notional machine?

A preliminary version of the results was presented at the 44th Annual Meeting of the Cognitive Science Society (Chiarelli and Muldner, 2022b).

5.1 Learning Outcomes

As the first step in measuring learning, the tests were graded. The first four test questions addressed variables and input and output and were categorized as “easy” questions for this analysis and the remaining three questions which additionally
included while loops with conditions were categorized as “hard” questions (comparing success on looping and non-looping questions is standard e.g., Mayer, 1976).

A grading rubric was developed and used to grade the tests. Part marks were possible for all questions except for the one multiple-choice question (Question 5, see Appendix B.3). The grading scheme is in Appendix B.3.

The control group consisted of 24 participants (15 had some programming experience from high school or at most one university level course while nine had no experience). The experimental group consisted of 24 participants (four had some programming experience from high school or at most one university level course and 20 had no programming experience). Two participants were excluded from the analysis. One participant in the experimental condition was excluded because they reported having completed more than one university programming course. To avoid ceiling effects, participants were excluded if they scored 90% or higher on the pretest (one participant in the control condition). Data from the other 46 participants (control $N = 23$, experimental $N = 23$) remained. We checked for outliers in the dependent variables used in the analysis but only report on them if they were present.

Descriptives for the pretest and posttest are in Table 5.1 and shown graphically in Figure 5.1. Code-tracing skill reflects mental model quality (Bayman and Mayer, 1988), and so pretest and posttest scores indicate the accuracy of mental models. On average, pretest scores were low for both groups but by chance, the average experimental group pretest score (19.57%) was lower than that of the control group (35.22%). Note that pretest scores were not all zero because participants with up to one university course of programming experience were eligible to participate. Average posttest scores for both conditions were around the mid 60% range and so higher than pretest scores, descriptively showing that students learned in both conditions. The average posttest
Table 5.1: Descriptives for the Pretest and Posttest, per Condition.

<table>
<thead>
<tr>
<th></th>
<th>Pretest (%)</th>
<th>Posttest (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>SD</td>
</tr>
<tr>
<td>Control</td>
<td>35.22</td>
<td>24.84</td>
</tr>
<tr>
<td>Experimental</td>
<td>19.57</td>
<td>21.47</td>
</tr>
</tbody>
</table>

Figure 5.1: Mean Pretest Scores (%) and Mean Posttest Scores (%), by Condition.

score for the control group (68.91%) was slightly higher than the average for the experimental group (61.96%), but the control group also started with higher pretest scores.

Overall, without considering condition, participants did learn from the programming lesson, as reported by a paired samples t-test, $t(45) = 10.39$, $p < 0.001$, $d = 1.53$.

Of primary interest was the impact of condition on learning, and, in particular, whether notional machine instruction in the experimental condition improved learning
over standard instruction in the control condition.

We first checked for a priori group differences in pretest scores. An independent samples t-test with condition as the independent variable and pretest scores as the dependent variable reported a significant effect of condition, \( t(44) = 2.29, \ p = 0.027, \ d = 0.67 \), with the control group’s pretest scores significantly higher than the experimental group.\(^1\)

To account for the a priori difference between conditions, we used a normalized gain formula to measure learning:

\[
\text{normalized gain} = \frac{\text{posttest score}\% - \text{pretest score}\%}{100 - \text{pretest score}\%}
\]

Normalizing gains is a standard approach when analyzing data (e.g., Abbasi et al., 2021).

To determine if the data contained outliers, we calculated the interquartile range with respect to normalized gain per condition. Outliers were defined as scores below 25\(^{th}\) percentile \(-1.5 \times IQR\) or above 75\(^{th}\) percentile \(+1.5 \times IQR\). There were no outliers in the experimental condition. In the control condition one participant with an unusually low normalized gain score (-75%) was identified as an outlier.\(^2\) The outlier was removed from subsequent analysis, leaving 45 participants (control \( n = 22 \), experimental \( n = 23 \)).\(^3\)

The average normalized gain per condition is in Table 5.2 and shown graphically

\(^{1}\)Removing data from participants who did not learn from the lesson resulted in no significant pretest difference between conditions, \( t(41) = 1.50, \ p = 0.140, \ d = 0.46 \). However, to avoid excluding relevant data, these data points were not removed from analysis.

\(^{2}\)For the control condition, the IQR was 53.77% and the 25\(^{th}\) percentile was 21.88%. 21.88%\(-1.5 \times 53.77\% = -58.786\) and the participant’s normalized gain score of -75% was far below this boundary.

\(^{3}\)This outlier was not influential, i.e., did not impact overall pattern of results (see the following analyses).
Table 5.2: Normalized Gain, per Condition.

<table>
<thead>
<tr>
<th></th>
<th>Normalized Gain (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M$</td>
</tr>
<tr>
<td>Control</td>
<td>50.58</td>
</tr>
<tr>
<td>Experimental</td>
<td>52.89</td>
</tr>
</tbody>
</table>

*Note.* This table is based on normalized gains and not the raw difference in pretest and posttest scores.

Figure 5.2: Mean Normalized Gain (%), by Condition.

*Note.* This graph is based on normalized gains and not the raw difference in pretest and posttest scores.

in Figure 5.2. On average, the normalized gain was similar for both groups.\(^4\)

An independent t-test with condition as the independent variable and the normalized gain score as the dependent variable reported no significant difference in gain scores between conditions, $t(43) = 0.25$, $p = 0.801$, $d = 0.08$.\(^5\)

\(^4\)Removing the outlier slightly increased mean normalized gain from 45.12% to 50.58% for the control group.

\(^5\)Including the normalized gain outlier did not change the pattern of results, i.e., they were still $ns$, $t(44) = 0.75$, $p = 0.461$, $d = 0.22$. 
Table 5.3: Descriptives for Scores on Easy and Hard Questions, per Condition.

<table>
<thead>
<tr>
<th></th>
<th>Pretest Easy</th>
<th>Posttest Easy</th>
<th>Pretest Hard</th>
<th>Posttest Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M$</td>
<td>$SD$</td>
<td>$M$</td>
<td>$SD$</td>
</tr>
<tr>
<td>Control</td>
<td>52.72</td>
<td>27.43</td>
<td>80.44</td>
<td>18.01</td>
</tr>
<tr>
<td>Experimental</td>
<td>35.87</td>
<td>36.60</td>
<td>71.20</td>
<td>24.55</td>
</tr>
</tbody>
</table>

5.2 Learning on Easy Versus Hard Questions

Analyzing gains based on question difficulty provides more sensitivity for identifying learning differences. For instance, in some prior work, conditional effects only emerged for difficult questions but not easy ones (Muldner et al., 2014). Distinguishing between performance on non-looping/easy questions and looping/hard questions permits analysis of how the learning condition relates to the type of question (as was done in Mayer, 1975, 1976).

We separated the pretest and posttest data by question type (easy or hard, see Table 5.3). Descriptively, on average the control group had higher easy and hard pretest and posttest scores compared to the experimental group. For both groups, the average scores on easy and hard questions were higher at posttest than at pretest showing that participants in both groups learned for both types of questions. Pretest scores on easy questions were higher than on hard questions. A paired t-test comparing pretest easy scores and pretest hard scores reported a significant difference, $t(45) = 6.93$, $p < 0.001$, $d = 1.02$, confirming that easy questions were in fact easier for participants initially.

The present analysis mirrored the steps for the overall analysis reported above. We first checked for a priori differences by conducting an independent samples t-test with the condition as the independent variable and the easy question pretest scores as
Table 5.4: Descriptives for Normalized Gain Scores for Easy and Hard Questions, per Condition.

<table>
<thead>
<tr>
<th></th>
<th>Normalized Gain Easy</th>
<th></th>
<th>Normalized Gain Hard</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>SD</td>
<td>M</td>
<td>SD</td>
</tr>
<tr>
<td>Control</td>
<td>50.76</td>
<td>45.27</td>
<td>35.19</td>
<td>58.76</td>
</tr>
<tr>
<td>Experimental</td>
<td>43.33</td>
<td>56.77</td>
<td>39.14</td>
<td>25.46</td>
</tr>
</tbody>
</table>

the dependent variable. There was no significant difference on pretest easy questions between groups, \( t(40.79) = 1.77, p = 0.085, d = 0.52 \). However, the control group hard question scores were significantly higher on these questions than the experimental group scores, \( t(33.08) = 2.23, p = 0.033, d = 0.66 \).

As above, learning was operationalized using normalized gain. Four participants were at ceiling for the pretest easy questions and so were removed from analysis (\( n = 20 \) experimental and \( n = 22 \) control participants remained). The average normalized gain scores per condition are in Table 5.4.

To check for outliers, we calculated the interquartile range with respect to normalized gain on hard questions per condition and on easy questions per condition. There was one outlier in the experimental condition for normalized gain on easy questions with an unusually low score of -100%. The outlier was removed from analysis (\( n = 19 \) experimental participants remained).\(^6\) In the control condition the same participant that was an outlier for normalized gain overall was an outlier for normalized gain on hard questions with an unusually low hard question normalized gain score of -150%. That participant was removed (\( n = 21 \) control participants remained).\(^7\)

\(^6\)For the experimental condition on easy questions, the IQR was 62.50% and the 25\(^{th}\) percentile was 15.63%. 15.63%−1.5 × 62.50% = −78.12 and the participant’s normalized gain score of -100% was far below this boundary.

\(^7\)For the control condition on hard questions, the IQR was 60.83% and the 25\(^{th}\) percentile was 16.67%. 16.67%−1.5 × 60.83% = −74.58 and the participant’s normalized gain score of -150% was far below this boundary.
On average, the normalized gain on easy questions was higher for the control group than for the experimental group, but this was not significant, $t(38) = 0.16$, $p = 0.875$, $d = 0.05$.\footnote{With the outlier, this result remains \textit{ns}, $t(40) = 0.47$, $p = 0.640$, $d = 0.15$.} For hard questions, the experimental group’s normalized gain was slightly higher than the control group, but this was not significant, $t(33.29) = 0.35$, $p = 0.729$, $d = 0.11$.\footnote{With the outlier this result remains \textit{ns}, $t(21.19) = 0.29$, $p = 0.776$, $d = 0.86$.}

### 5.3 Exploratory Analysis of Self-Explanations

Comparing the content of self-explanations between groups can provide insight into participants’ understanding of the lesson. In general, analyzing the content of self-explanations requires qualitative analysis. A relevant analysis corresponds to extraction of mental models from students’ self-explanations. To illustrate, to compare mental models of participants learning about the circulatory system, Chi et al. (1994) first developed six mental models of the circulatory system with varying accuracy. Then, for each participant, they extracted the participant’s mental model based on a detailed analysis of the content of posttest self-explanations, and categorized it as one of the six mental models that they had developed. This process took months\footnote{Personal communication between Chi and Muldner.} but shed light on participants’ mental models of the circulatory system. A similar analysis for our participants would allow us to compare self-explanations and mental models between conditions and could explain the lack of a learning difference between groups. For instance, if participants in both groups have mental models that correctly represent memory, then the notional machine is not more beneficial than a code-trace table for that concept. Thus, formal qualitative analysis would be informative in future work.
but is beyond the scope of the current thesis. However, we were curious whether there were any indications in the explanations that both groups understood the notional machine similarly (since mental models of the notional machine can be developed via implicit instruction, e.g., Hertz and Jump, 2013, or explicit instruction, e.g., Mayer, 1975). To this end, here we present anecdotal observations of self-explanations in response to two prompts (prompt 2 and prompt 4) within the tutoring system and oral responses to question 7 on the posttest, and discuss the similarity of the explanation features between the groups. To do this, we extracted participant self-explanations from the tutor. Tutor logs from six participants were lost due to an error on the DataShop site. Logs from 41 participants (control \( n = 21 \), experimental \( n = 20 \)) remained. Prompts were identical for both groups.

### 5.3.1 Self-Explanations for a Code Trace

The second prompt screen in the tutor (see Figure 5.3) prompted participants to describe how the two-line program runs by entering a text explanation for each line of code. For the experimental group, 13 participant responses made explicit reference to the notional machine (e.g., “The pointer reads line 1, then sets a value of 0 to the variable response which is put into the program’s memory. The program pointer is set on line two, where it reads the right hand side first, it puts the value 0 in place of the variable and adds 0+1, which equals 1. This is then added to the memory of the program which updates it to a new value”) while the remaining eight responses did not (e.g., “response is assigned by 0; response is added by 1”). While, as expected, the control group did not mention the notional machine, responses were similar to the latter type of experimental group responses (e.g., “here response value is 0; here response value is updated to 1(0+1)”). So, anecdotally, the content of participants’
What happens when this program is run?

Explain what happens at each line of the program.

1. \texttt{response = 0}

2. \texttt{response = response + 1}

Figure 5.3: Prompt 2 in the Tutor.

Self-explanations of a code trace were similar between groups, suggesting that both groups were similarly capable of self-explaining a code trace.

5.3.2 Self-Explanations for a Programming Concept (While Loops)

Prompt 4 (see Figure 5.4) asks for a high-level conceptual explanation of while loops. Eight participants in the experimental group referred to the notional machine in their response (e.g., \textit{The While loop sets a condition and uses indents to follow after if the condition is true, it will repeat these steps and check if the condition is true and create a loop until the condition is false. which is where the computer's pointer moves out of the loop and read the unindented line}); the other 12 participants did not. Fewer participants referred to the notional machine for this prompt than prompt 2, perhaps because participants found the notional machine to be more relevant when prompted for a code trace (and monitoring memory, input/output etc., as was the case for prompt 2). So, again, experimental group self-explanations were similar to
self-explanations from the control group (e.g., a self-explanation from the control group: “While loops are a way of repeating programs continuously until some condition is met to stop the loop.”).

5.3.3 Posttest Self-Explanation Prompt

Question 7 on the posttest asked participants to code trace the program in Figure 5.5, assuming the user types in *king* and then *queen*. A correct mental model of the notional machine would track variables and their values in lines 1-3, then repeat execution of lines 4-6 (by tracking output, input, and variable values) until the condition on line 4 is met, and then execute lines 7-9 by updating variable values and tracking output. A code trace detailing the variable values, input, output, and flow of execution indicates a correct mental model of the notional machine.

Evidence of correct mental models of the notional machine based on code-tracing accuracy was found in participants’ answers from both groups (see Tables 5.5 and 5.6 for examples). Some experimental participants explicitly incorporated notional machine components into their mental models (e.g., Table 5.6) but accurate mental models
5.3.4 Common Misconceptions

We now discuss some common misconceptions in participants’ mental models, illustrating these with examples from self-explanations of the program in Figure 5.5 (this program was on the posttest). Misconceptions were present in both groups’ self-explanations (see Table 5.7). The misconceptions point to areas in the lesson where participants might not have received enough support to represent the concepts accurately in their mental models.

The indentation misconception. Participants with this misconception expect code to repeat even when the code is outside of a while loop. For example, an unindented print statement directly under a while loop should not print at each iteration of the loop. A correct mental model would represent that while loops in Python include only the indented lines of code and would, therefore, only simulate repeating the indented code lines when the condition of the while loop is true. See examples 1 and 2 in Table 5.7, for code traces with this misconception.

The lesson explained that the indented lines are part of the loop and the unindented
Response from control participant (C23).

“So, the first thing that it does is that it assigns the string “queen” uh to the variable answer.
And then after that it will it’s basically setting up variables for user and counter as well except the string “...” is for user and the value 0 is for counter.
Then line 4 it begins a loop, a while loop basically and uh the condition that it uh that it has to be met is that it checks if the user is not equal to answer so the first time around uh this will be true so that means that it will execute the loop since uh “...” is not equal to “queen” in that case
And so, uh line 5 then it will print try again because it’s not correct, so it prints out once.
And then line 6 it will ask the user to guess it again, then it will answer then they will answer “king”.
Then it will see that 4 is true again.
And so, it will execute the loop once more.
And so, it will print try again one more time.
And then it’ll go to line 6 and it will ask the user one more time to guess the card and then and then they’ll answer “queen”.
And then finally uh line 4 it will see that it’s not true because it’s uh it isn’t it’s equal to answer this time which is “queen” So, then it’ll skip uh the loop it will break the loop basically
And then it’ll go to line 7 counter is increased by 1 so right now counter is just 1.
And then line 8 it will just print the string woohoo.
And then 9 finally it will it’ll print the value of counter which is just 1.”

This code trace simulates all of the necessary details of program execution. It simulates the correct flow of execution, executing unindented lines once and executing the while loop until the condition is met. It monitors variables and their values as they are initialized and updated and it tracks output and input.
Table 5.6: Example of Oral Response to Question 7 Demonstrating that the Experimental Participant has Formed an Accurate Mental Model of the Notional Machine.

**Response from experimental participant (E2).** Explicit reference to notional machine is in **bold**

"Okay so the **program pointer** is set on line 1 and so when the computer, like, reads the line, it will enter a new **memory** where it has a variable **answer** and assigns the value “queen” to it. And then it will move on to line 2 where it will have a new **memory** with a new variable called **user** and assign, you know, the “…” onto it. And then it will move to line 3 where it will have another, uh, variable called counter and assign the value 0. And then it will move on to line 4 where it will compare the value of **user** and, um, if the statement is true that user does not equal the answer which is “queen”. The, the program will move on to line 5 which will print on the **output** screen, uh, the string try again And then it will move to, to line 6 and, uh, the user will **input**, um, the, no, the program will, it will show on the screen the text guess the card and the user can then **input** “king”. And then it will move back to, it will move back to, uh line 4 where it will see if the statement is true or not, and the user placed “king” onto the, wait yeah, like it’ll compare the answers, so “king” does not equal “queen”. And it’ll move to line 5 where it will uh print on the **output** screen try again. Move to line 6 where the user can then **input** the word “queen” when it says guess the card. And then it will move back to line 4 and it will see that the user has like the same answer so, yeah so then the statement is false that “queen” does not equal the answer, but the answer is “queen”. So then it will move to line 7, and line 7 the **pointer** will, like the program will read line 7 and then it will say, um the counter + 1 so it’s telling use to - it’ll increase the counter by a value of 1 and then the counter, it’ll be 1 And then so it moves to the line 6 where it’ll print on the **output** screen the string woohoo. And then on line 9, it will print the counter value."

This code trace accurately simulates program execution. It traces the correct flow of execution with unindented lines executing once and with the while loop repeating until the condition is met. It explicitly tracks variables and their values as they are initialized and updated and it tracks output and input throughout the program.
<table>
<thead>
<tr>
<th>Misconception</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 The indentation misconception (C31)</td>
<td>“[…] it is true so it will do another iteration of the while loop. And the counter will add 1 um to that so it will now the counter will be equal to 1 after entering “king” […]” A correct code trace would not increase the counter until after the while loop, i.e., the counter would still have a value of 0 after the user types in “king”.</td>
</tr>
<tr>
<td>2 The indentation misconception (E18)</td>
<td>“[…] We input “queen” and this time we add 1 more to the counter because before it was already 1 so now it’s 2 […]” A correct code trace would not increase the counter until after the while loop, i.e., the counter would still have a value of 0 after the user types in “queen”.</td>
</tr>
<tr>
<td>3 The input misconception (C25)</td>
<td>“[…] And then line 2 um the user input “king” […]” A correct code trace would initialize the value of user to “…”</td>
</tr>
<tr>
<td>4 The input misconception (E20)</td>
<td>“[…] So line 2 prompts the user using the computer to input an answer um, so the user then inputs um “king” […]” A correct code trace would initialize the value of user to “…”</td>
</tr>
<tr>
<td>5 The pattern misconception (E28)</td>
<td>“[…] So when the computer sees that you’ve entered the wrong username value or whatnot then it goes back to verify it and then comes back again to tell to tell you to re-enter it in the program which you do and then it checks to see if it matches up with the correct response and if it does get that and it matches then you are able to to enter into the you’re welcomed into the the program […]” A correct code trace would provide line-by-line details, monitoring input, output, variables and their values.</td>
</tr>
<tr>
<td>6 The execution order misconception (C27)</td>
<td>“[…] And let’s say that on his second try the user puts the right answer so “queen” as a response um he would put it in line 2 this will make him go directly from line 4 um to line 7 […]” A correct code trace would not jump back to lines before the while loop while simulating while loop execution.</td>
</tr>
</tbody>
</table>
lines are not, and further emphasized this point with a self-explanation prompt asking which lines of code were inside a while loop in a sample program. The lesson demonstrated the code trace of a while loop with the counter inside of the loop, so participants were familiar with that pattern. Some participants with the indentation misconception overapplied the familiar pattern of a repeatedly increasing counter to a program on the posttest that had a counter outside of a while loop instead. The program pointer component of the notional machine did not explicitly represent the beginning and end of while loops beyond code indentation (for instance by containing those lines of code in a block) which seems to have left this concept open for misrepresentation in participant mental models. Future work could examine ways of making explicit which lines of code lie within the loop in the context of the notional machine.

The input misconception. With this misconception, participants expect the program to be asking for user input when the input function is not present. For example, some users thought that a variable named “user” would get user input even when there was no input function in the code. A correct mental model would represent that user input is received only when the input function is used. See examples 3 and 4 in Table 5.7, for code traces with this misconception.

The pattern misconception. With this misconception, participants recognize general high-level patterns in a program but do not trace the program’s details. For example, when simulating program execution, a correct mental model would track the details of variables, input, output and flow of execution while code tracing. See example 5 in Table 5.7, for a code trace with this misconception.

The execution order misconception. With this misconception, participants do not accurately simulate the order in which code lines are executed. For example, for the
content covered in our introductory lesson, a code trace should not return to previously executed lines of code unless they are part of a loop. A correct mental model would accurately simulate the flow of execution when tracing programs. See example 6 in Table 5.7, for a code trace with this misconception.

In sum, there was overlap in self-explanation content between the two groups. Some participants in both the control and experimental groups demonstrated accurate mental models of the notional machine based on code-traces (see Tables 5.5 and 5.6). Some participants in both groups had misconceptions about program execution (see Table 5.7), suggesting that the notional machine may not have provided support for those concepts superior to that of a code-trace table.
Chapter 6

Discussion

Based on prior results (Mayer, 1989), we predicted that the experimental group would learn more than the control group. We did not find evidence for this prediction: there was no significant difference in learning between groups. Posttest scores were not at floor or ceiling, so the lack of a significant difference is not a result of the lesson or tests being too challenging or too simple. We discuss the results and future work below.

Our study was inspired by prior research (Mayer, 1975, 1976; Bayman and Mayer, 1988). Mayer (1975) compared learning from different versions of instructional materials that included the notional machine or not. Experiment 1 showed that groups learning from notional machine materials performed better on posttest code-tracing questions, while groups without notional machine instruction performed better on code generation questions. In the second experiment, groups given notional machine materials again performed better on code-tracing questions but worse on code generation questions, a pattern similar to that of experiment 1. Our tests only included code-tracing questions so these interactions could not be tested.
Bayman and Mayer (1988) compared learning from different types of lessons including lessons with notional machine instruction. These results separated high prior knowledge students from low prior knowledge students (based on math knowledge). While our study did not collect data on prior math experience, we did recruit participants with limited programming knowledge, and so our participants may have been similar to Bayman and Mayer’s low prior knowledge group. In Bayman and Mayer’s study, the low prior knowledge group’s code-tracing performance was similar in the notional machine and other non-standard instruction groups (the latter were still given some code-tracing instruction). In our study, a similar pattern emerged (i.e., code-tracing performance was similar for participants learning from a notional machine lesson vs. standard code-tracing lesson).

Mayer (1976) tested when to provide students with the notional machine model. Groups presented with the model at the very start of instruction performed significantly better overall than other groups. Moreover, students learning from a text that included information on the notional machine performed better on harder questions involving code-tracing programs with loops than students learning from a ‘rule-based’ text without the model, but the rule-based condition did better on easier questions about individual functions (e.g., about the \texttt{input()} function not within the context of a larger program) than the notional machine text condition. In our study, the experimental group was provided with notional machine instruction right from the start (this is similar to Mayer’s notional machine “before” groups), while the control group did not have it at all. The experimental group’s normalized gain on hard (loop) questions (39.14%) was slightly higher than the control group’s normalized gain (35.19%), but not significantly so. In contrast, for the easy questions, the control group’s normalized gain (50.76%) was higher than the experimental group’s normalized gain (43.33%), but
this was not significant. Thus, while the overall pattern of results is similar between
the two studies, this is only at the descriptive level.

In sum, Mayer (1975) found that notional machine groups performed better
on code tracing questions and other groups performed better on code-generation
questions. We did not assess code-generation performance so this interaction could
not be tested. Bayman and Mayer (1988) found that low prior knowledge students
performed similarly when learning from code-tracing instruction and notional machine
instruction. This matches our result of similar gains from experimental and control
groups. Mayer (1976) found that participants introduced to a notional machine prior
to other instructional materials performed significantly better than other groups. We
did not find a significant learning difference between the two groups in our study.
While our study was informed by this prior work, there were differences, for instance
related to the population and instructional materials. We next discuss reasons for
these differences and how they potentially influenced the results.

6.1 Populations

In Mayer (1975), Mayer (1976) and Bayman and Mayer (1988), students were only
eligible to participate if they had no programming background. In contrast, in our
study, participants were eligible if they had some programming background (up to
one university course). It may be that students with some programming experience
already formed mental models for code tracing, meaning that the notional machine
was presented too late (after their first programming lesson) to guide mental model
formation or influence learning more than standard code-tracing instruction. We
checked for this possibility by comparing normalized gain for participants with no prior
experience in both groups. The average normalized gain for experimental participants without prior experience \((M = 50.23\%, SD = 23.81\%)\) was higher than for the control participants without prior experience \((M = 41.75\%, SD = 21.52\%)\). This difference in average normalized gain between groups (8.48%) is descriptively larger than the difference in average normalized gain between groups when considering participants of all eligible experience levels (2.31%). The pattern remained the same, i.e., the difference in average normalized gains between groups was still not significant, \(t(27) = 0.91, p = 0.370, d = 0.37\). However, when we included only participants without any prior experience, the effect size increased (from \(d = 0.08\) to \(d = 0.37\)), so the lack of significance may be a power issue given that the sample size was reduced when we only included participants without prior programming experience.

6.2 Instructional Materials

6.2.1 Video Lessons

Our instructional materials shared similarities with, but were not identical to, ones used in prior work (Mayer, 1975, 1976; Bayman and Mayer, 1988). These differences could have affected our experiment’s results. The notional machine in Mayer (1975), Mayer (1976) and Bayman and Mayer (1988) was presented statically on paper rather than ‘dynamically’ in video lessons. We used dynamic materials based on feedback from our pilot. In particular, videos permitted highlighting changes to the notional machine during a code trace by updating the notional machine instead of by presenting a sequence of static notional machine states, which confused some participants in our pilot. Our control group also received video lessons instead of text-based lessons. It is possible that the dynamic video lessons made it easier for all participants to
follow the code trace than when the lesson was text-based. For example, participants reading a lesson might skip over written explanations of what happens each time a while loop executes but our participants did not skip over sections of the videos (this was monitored via Zoom screen sharing). Observing the details of a dynamic code trace could have been more influential than the method used to code trace, making learning similar for both groups.

6.2.2 Notional Machine vs. Code-Trace Table

A notional machine includes abstractions of key computer components necessary for a code trace and can be used to simulate their interaction during program execution. For example, using the notional machine to code trace a line of code that sets a variable equal to some user input will represent how that line of code involves output to prompt the user, input to receive user input, and memory to store the variable and its value. A standard code-trace table also stores variables and their values (e.g., Hertz and Jump, 2013; Xie et al., 2018) but does not explicitly incorporate all computer components and interactions necessary to simulate program execution like the notional machine does (e.g., see code-trace table in Figure 4.8). Our control group learned to code trace with a standard code-trace table. To demonstrate code tracing, the control group’s video lessons dynamically updated the code-trace table to include variables and to monitor their values. This differs from Mayer (1975), Mayer (1976) and Bayman and Mayer (1988) non-notional machine groups. For these groups, code-tracing instruction explained the lines of code and execution in English sentences only. So, for instance, the explanations verbally described how the values of variables were updated throughout a code trace but did not visually represent this in a table. A table to keep track of a code trace is beneficial (Cunningham et al., 2017). Since
our control and experimental group had similar learning outcomes, a code-trace table may have been sufficient support for learning to code trace.

6.2.3 Self-Explanation Prompts

Mayer (1975), Mayer (1976) and Bayman and Mayer (1988) did not incorporate self-explanation prompts (although Mayer (1975) did include practice questions). Prior work has shown that students who self-explain instructional material learn more than those who do not (Chi et al., 1989). Prompting for self-explanation encourages constructive engagement with the material and increases learning (Chi et al., 1994; Bisra et al., 2018). We included self-explanation prompts in both versions of the tutor (control and experimental) to promote engagement with the material for all participants. Participants did answer the prompts suggesting at least some level of engagement. To determine this, we checked the number of words produced per self-explanation. Both groups produced on average about 12-16 words per self-explanation. The control group had slightly shorter self-explanations ($M = 11.65, SD = 5.23$) than the experimental group ($M = 15.89, SD = 8.50$), $t(28.71) = 2.15, p = 0.040, d = 0.70$. However, this was likely because this group did not often include information on notional machine components in their explanations.

The self-explanation effect could have been more influential than the type of lesson (notional machine or standard), increasing constructive engagement with the lesson for both groups and leading to similar performance between groups. However, the posttest scores for both groups were on average around 60% - 70% so there was still opportunity for more learning.

\footnote{One experimental participant and one control participant were identified as outliers (above 75\textsuperscript{th} percentile $+ 1.5 \times$ IQR) and removed from this analysis. With the outliers, the trend was the same, $t(39) = 1.93, p = 0.061, d = 0.60$.}
6.3 Other Limitations and Future Work

We found large standard deviations in test scores (over 20% for both groups), indicating individual differences in performance. It is possible that the variety in participants’ backgrounds influenced results. Given that the sample size was not large \((N = 46)\), increasing the sample size could clarify if the condition affects learning (although this is unlikely since the difference in gains between groups was small). Controlling for prior knowledge more by only including participants without any programming experience, however, is more promising. As discussed above, the difference in learning between groups was larger when only analyzing participants without any prior programming experience versus all participants. Investigating learning from pure novices could address this limitation. Alternatively, future work could involve implementing a tutoring system that can adapt to support individual differences.

Participant engagement is another limitation on results, even in in-person studies. For example, Cunningham et al. (2019) found that, to save time, students often look for patterns or do not code trace. This may be even more of a limitation in an online context. For our study, we addressed this by incorporating self-explanation prompts to encourage engagement with the lesson and by monitoring participant progress via screensharing. Even so, cameras were turned off so it is possible that participants in both groups were not as focused as they would have been in a physical lab setting (e.g., they could be looking at their phone while the lesson videos played even though they were asked not to).

In our study, the posttest took place immediately after the lesson. Results from a delayed posttest could indicate whether the effects of learning persist (e.g., Mayer and Bromage, 1980) or change (e.g., Rittle-Johnson et al., 2017) after a delay. A delayed posttest was not feasible for this study because it requires participants to return for
follow up sessions within a fixed period of time, which is challenging in a lab study (e.g., Di Lonardo Burr et al. (2020) were unable to recruit sufficient participants for delayed posttest in their lab-based study).

As mentioned in the Results chapter, formal qualitative analysis of self-explanations and direct extraction of mental models from self-explanations were beyond the scope of this thesis. However, in future work these analyses could inform on mental models formed by the participants.

Our analyses produced null results for the research questions. Unfortunately, Null Hypothesis Significance Testing (NHST), the framework we used, does not allow for interpretation of null results. Bayesian statistics are becoming more common as they do allow for interpretation of null results, by quantifying evidence for the null and alternative hypotheses (Jarosz and Wiley, 2014). Subsequent analysis with Bayesian statistics could provide additional insight into our results.

6.4 Conclusion

To date, little work exists investigating the effect of notional machine instruction to help students learn the challenging skill of code tracing. This thesis investigated the utility of explicit instruction about the notional machine, in the context of a tutoring system that included video lessons about code tracing and self-explanation prompts. Participants learned from the tutor but the notional machine and standard instruction groups learned similarly overall and on easy and hard questions. To shed light on why, work is needed to identify participants’ mental models in both groups.
Bibliography


and Innovation Tracks, Practitioners’ and Doctoral Consortium, pages 419–423, Cham. Springer International Publishing.

Chiarelli, V. and Muldner, K. (2022b). Self-explaining the notional machine to improve novice programmers’ learning and mental models, supported by a computer tutor system. Abstract and poster presentation.


Appendix A

Pilot Materials

A.1 Pilot Consent Form
Informed Consent Form

Name and Contact Information of Researchers:
   Kasia Muldner, Carleton University, Department of Cognitive Science  
      Email: Kasia.Muldner@carleton.ca  
   Veronica Chiarelli, Carleton University, Department of Cognitive Science  
      Email: VeronicaChiarelli@cmail.carleton.ca

Project Title
Educational technologies & student learning (D)

Project Sponsor and Funder (if any)
NSERC Discovery grant

Carleton University Project Clearance
This study has received clearance by the Carleton University Research Ethics Board B (Clearance #111476).

Invitation
You are invited to participate in our online research study. This study involves filling in a brief questionnaire about programming, studying a programming lesson, and answering another questionnaire. To be eligible you cannot have taken more than one university-level class.

The information in this form is intended to help you understand what we are asking of you so that you can decide whether you agree to participate in this study. Your participation in this study is voluntary, and a decision not to participate will not be used against you in any way. As you read this form, and decide whether to participate, please ask all the questions you might have, take whatever time you need, and consult with others as you wish.

What is the purpose of the study?
The purpose of this study is to analyze how you interact with a tutorial that teaches you how to predict the output of programs.

What will I be asked to do?
The study is conducted online using the Zoom platform. If you agree to take part in the study, we will ask you to fill out some questionnaires on programming – you are not expected to have much background and it’s okay if you don’t know the answers to it. You will work virtually and the researcher will be online with you during the session. We will ask you to study a lesson that teaches about computer programming. Because we want to analyze the usability of our lesson design, we will ask you to think aloud during the study, which means verbalizing your thoughts as you work with the lesson; we may also ask you questions on your impressions of the lesson. We will use Zoom to audio record the session and the screen (your face will not be recorded).
This in-session data (audio and chat transcription) will be stored locally on the researcher’s computer. Operation data, such as meeting and performance data, will be stored and protected by Zoom on servers located in North America, but may be disclosed via a court order or data breach. The audio and screen-recording data will NOT be shared or publicly used and will be destroyed as soon as it is transcribed and analyzed. Please note that audio and screen recording is part of the data collection for this study, and so is a condition of consenting to participate. After the instructional activity, you will fill out a questionnaire similar to the one from the beginning. The study will take no more than 2 hours.

**Risks and Inconveniences**
We do not anticipate any risks to participating in this study.

**Possible Benefits**
You may not receive any direct benefit from your participation in this study. However, your participation may allow researchers to better understand how to design instructional materials to support student learning of programming.

**Compensation/Incentives**
As a token of appreciation, you will receive 2% extra credit for CGSC 1001 or $25 (up to you but you must be enrolled in CGSC1001 for the credit). If you withdraw before finishing the experiment, the amount of compensation will be prorated by time (e.g. if you withdraw halfway through the study, you will receive half the compensation).

**No waiver of your rights**
By signing this form, you are not waiving any rights or releasing the researchers from any liability.

**Withdrawing from the study**
During the study session, you have the right to end your participation for any reason, by stating that you do not want to continue. If you withdraw from the study, all information you have provided will be immediately destroyed (including the audio recording). Because we do not store any identifying information with the data, once you leave the session, withdrawal is not possible.

**Confidentiality**
We will treat your personal information as confidential, although absolute privacy cannot be guaranteed. No information that discloses your identity will be released or published. Research records may be accessed by the Carleton University Research Ethics Board in order to ensure continuing ethics compliance.
You will be assigned a code so that your identity will not be directly associated with the data you have provided. However, if you are being granted course credit for taking part in the study, identifying information will be retained by the SONA system using a code until the course credit is granted.
We will password protect any research data that we store or transfer. Research data will only be accessible by the researchers and the research supervisor and will not have any identifiable information. Thus, your name will not appear in any publications or other venues. Once the project is completed, research data will be kept and potentially used for other research projects on
this same topic (but as noted above, audio and research data will be destroyed as soon as it is transcribed/analyzed).

Data Retention
After the study is completed, your de-identified data will be retained for future research use (note that the audio and screen recordings will be destroyed once they are transcribed by the research team).

New information during the study
In the event that any changes could affect your decision to continue participating in this study, you will be promptly informed.

Ethics review
This project was reviewed and cleared by the Carleton University Research Ethics Board B. If you have any ethical concerns with the study, please contact Carleton University Research Ethics Board (by phone at 613-520-2600 ext. 4085 CUREB B or by email at ethics@carleton.ca).

Statement of consent – print name
I voluntarily agree to participate in this study. ___Yes ___No

Signature of participant (typed name) Date

__________________________    _____________________
Signature of researcher (typed name)      Date
A.2 Pilot Pretest

Question 1

What does the program print?

```python
result = 1
result = result + 10
print( result )
```

Question 2

User types “kat” - what does the program print?

```python
name = input("enter your name: ")
print(" you entered", name )
```

Question 3

User types “Z” - what does the program print?

```python
char = "K"
letter = input("Enter a letter: ")
res = letter + char
print ( res )
```

Question 4

When answering the questions, the user types “Ottawa”, “TO” - what does the program print? Show your work.

```python
result = 0
name = input("enter a city: ")
result = result + 1

name = input("enter another city: ")
result = result + 1

print( result )
print( name )
```
Question 5

What does the program print? Please show your work.

```python
result = 1
while result < 4:
    result = result + 1
    print(result)
print("he")
print(result)
```

Question 6

The user types “jack”, “king” - what does the program print? Please show your work.

```python
answer = "king"
user = "..."
while user != answer:
    user = input("Guess the card: ")
    print("try again")
print("hi")
```

Question 7

What does the program print? Please show your work.

```python
counter = 1
result = 0
while counter < 3:
    result = result + counter
    counter = counter + 1
print(counter)
print(result)
```

Question 8
The user types “613”, “902” - what does the program print? Please show your work.

```python
value = "613"
counter = 3
while value == "613":
    counter = counter - 1
    value = input("enter area code: ")
    print(counter)
print("You entered this many values:")
print(counter)
```

A.3 Pilot Lesson
Introduction to Programming

What is a program and why do we write programs?
A program is a series of instructions that specify the steps the computer will take to perform the task in the program. The instructions are expressed as code written in a programming language. This lesson will use the programming language Python 3. Why do we bother writing programs? One reason is to automate tedious tasks (e.g., count the number of passwords that are too short and/or easy to guess in a password file); another is to discover things about the world through machine learning; yet another reason is to build interactive systems that communicate with the user.

High-level model of the computer
Once a program is written, in order for the program to actually do something, it has to be “run”. To do that, the programmer hits the “run program” button, which tells the computer to read the program and carry out the instructions in it. The diagram below presents a high-level model of how a computer runs a program. We will use this model to simulate the computer’s actions so that we can predict what a program does.

High-Level Model of how the computer works when it runs a program

Let’s go through the model components.

Input: Programs can ask the user for information, which the user inputs at the program’s prompts. For instance, a program could help the user with an online order (e.g., by asking them to provide the product name, shipping address, credit card information). In the model above, whenever the user provides input, we will add it to the Input box (box at bottom left), to help us keep track of what information the computer was given by the user.

Memory: A computer has a memory, which keeps track of all of the items used by the program (e.g., if a user enters some data, that data will be added to the computer’s memory). The memory keeps track of all of the items used by the program and the values of those items. As the program runs, new information is added to the computer’s memory when the program introduces a new item. Information in memory can also be updated, when the value for an existing item in memory changes (see Memory area at the top) – how this occurs will be explained shortly.

Program pointer: When a computer runs a program, it does so in a systematic way. Starting at the first line of the code, it will execute the instructions line-by-line. The computer needs to keep track of which line it is currently reading in the program. This is represented in the model by an arrow that keeps track of the current program line being read (see the Program Pointer box in the bottom middle of the model, notice that it is pointing to line 1 of the program to start). We will show how to use this model feature soon.

Output: Programs produce output (e.g., tell the user how much the online order cost). This is represented in the model with the output window (see the Output box at the bottom right).

At this point, how do you think the computer runs a program?
Programing basics: Variables

The fact that people have names (e.g., “Mary”) makes it easy to refer to them (e.g., “Mary is sitting in the front row”). In programs, there are no “people” (obviously) but there is data. This data is labelled by variables. Variables are super important because they allow the program to refer to data and perform operations on it.

To create a variable in a Python program, we need to:
- Write down the variable name (what we call it is up to us)
- Assign it a value using the assignment operator “=”

Example 1: A one-line program that creates a variable and assigns a value to it:

```
counter = 0
```

Here is a little more information about the above one-line program:
- `counter` is not a special word – it is just a variable name. The programmer decides what to call a variable. Variable names can be any sequence of letters, numbers and underscores but must start with a letter or underscore and can’t contain spaces / other characters
- The assignment operator (=), however, is a special symbol. While this looks the same as an equal sign, we call it an assignment operator because it assigns a value (the thing to the right of the “=” to the variable name (the thing to the left of the “=”)
- Python is very picky about the way the program is written (i.e., the syntax used). To create a variable, you must use a specific syntax - here is a high-level template for it:

```
[variable_name] [assignment operator] [value]
```

What happens if we run the above one-line program? We gave the answer away above, but for the sake of completeness, let’s use our model (page 1) to show how a computer reads this program:

```
Program Pointer is set to line 1 of the program, so the following happens:
- Read Line 1: assign 0 to variable counter. This data is added to the first memory slot in the model, see top left
```

More examples of assignment to variables

Example 2: As we already saw, we can assign numbers to variables in a program:

```
expenses = 35
```

What happens when this program is “run”? The value 35, which is a number, is assigned to the variable expenses.
Example 3: We can assign a *string* to a variable (a *string* is a series of characters, with quotation marks around them):

```python
1 city = "ottawa"
```

Example 4: We can assign a *variable* to a second variable:

```python
1 counter = 0
2 new_counter = counter
```

What happens when this two-line program is “run”? The steps the computer takes are illustrated with the model below. When we use the model to simulate running a program, each step of the program is represented in its own model diagram. So, the first diagram shows a snapshot of the state of the computer after executing line 1 of the program and the second diagram shows the state of the computer after executing line 2 of the program.

- **[set program pointer to line 1]**
- **Read line 1:** Assign 0 to variable *counter*  
  [this results in the variable *counter* being added to a memory slot in the model and its value, see top left]

- **[move program pointer to line 2]**
- **Read line 2:**
  - Figure out the value of *counter* in line 2 – that value is 0
  - Assign that value (0) to the variable *new_counter* [this results in the variable *new_counter* being added to the next memory slot in the model and its value]

A caveat: the variable *counter* in line 2 had to have a value assigned to it before we could assign it to *new_counter*.

Example 5: We can assign the result of an operation to a variable:

```
1 expenses = 35
2 total = expenses + 10
```

What happens when this program is “run”? Here are the steps the computer takes (program shown on the left, steps on the right):

- **Read line 1:** Assign *expenses* to variable *expenses*
- **Read line 2:**
  - Evaluate the right hand side of “=“: (step1) *expenses* is 35, (step2) 35 + 10 is 45
  - Assign the value 45 to the variable *total*

A caveat: the variable *expenses* in line 2 had to have a value assigned to it before we could assign it to *total*.

**Note:** This example used the + operator (to add two numbers in the program) Python can also do calculations using – (subtraction), * (multiplication), / (division).
Example 6: We can change the value of a variable:

```
1 counter = 0
2 counter = 1
3 counter = counter + 5
```

What happens when this program is “run”? Here are the steps the computer takes:

- Read line 1: Assign 0 to variable `counter`
- Read line 2: Assign 1 to variable `counter` – this overwrites what was in `counter`, so it is now 1
- Read line 3:
  - Evaluate the right hand side of “=”: (step 1) `counter` is 1, (step 2) 1 + 5 is 6
  - Assign the value 6 to the variable `counter`, overwriting its previous value

Example 7: More than one variable on the right-hand-side of the assignment operator:

```
1 lego = 3
2 toy = 7
3 toy = toy + lego
```

What happens when this program is “run”? Here are the steps the computer takes:

- [set program pointer to line 1]
- Read line 1: Assign 3 to variable `lego`
  [this results in the variable `lego` being added to a memory slot in the model and its value, see top left]

- [move program pointer to line 2]
- Read line 2: Assign 7 to variable `toy`
  [this results in the variable `toy` being added to a memory slot in the model and its value]

- [move program pointer to line 3]
- Read line 3:
  - Evaluate the right hand side of “=”: (step 1) `toy` is 7, (step 2) `lego` is 3, (step 3) 3 + 7 is 10
  - Assign the value 10 to the variable `toy`, overwriting its previous value
    [update the model, see `toy` slot]
Programming basics: output and input

The print function shows the output of a program to the screen.

Example 8: Here is a simple program using print that prints hello to the screen:

```
1 print("hello")
```

Example 9: Programs can print the value of a variable. This program example prints mary to the screen (left):

```
1 person = "mary"
2 print(person)
```

What happens when this program is “run”? Here are the steps the computer takes:

- Read line 1: Assign "mary" to variable person
- Read line 2: print the value of the variable person to the screen

Takeaways:

- If there is a variable name in a print statement, the computer prints the value assigned to that variable
- When strings are printed, they are printed without quotation marks

The input function asks the user of the program for some input, i.e., information.

Example 10: A program that asks for user input:

```
1 password = input("what is your password?")
2 print("you entered")
3 print(password)
```

What happens when this program is “run”? Here are the steps the computer takes:

- [set program pointer to line 1]
- Read line 1: Ask the user for their password. The program waits at line 1 until the user types something. Let’s say the user typed “topsecret”. Assign “topsecret” to variable password.

  [this results in the variable password being added to a memory slot in the model and its value, see top left]

- [move program pointer to line 2]
- Read line 2: print to the screen you entered

  [this results output added to the model, see bottom right]
Putting it all together: Code tracing

Code tracing involves going through a program line by line to simulate what the computer does when it runs the program. While code tracing we need to keep track of the variables and their values, as well as the input and output of the program.

Example 11: Code tracing a program with variables, input, and output

Let’s code trace the program on the left (see the code trace on the right):

```
1. counter = 0
2. password = input("google password ")
3. counter = counter + 1
4. password = input("amazon password ")
5. counter = counter + 1
6. print(counter)
7. print(password)
```

What do you think the model would look like for the steps of this program?

- [move program pointer to line 3]
- Read line 3: print to the screen topsecret
  [this results new output added to the model, see bottom right]

To summarize, not including the user prompts, the program prints:

```
2
lollipop
```
Programming basics: While loops

There is often a need to repeat parts of a program until a condition is met (e.g., keep asking the user for a password until they get it right). One option to make this happen is to re-type the repeated program lines but that would be very inefficient. Luckily, Python provides an easy way to repeat the necessary code with a while loop. Here is an example of a program with a while loop:

```python
response = "--"
while response != "test123":
    response = input("Enter password: ")
    print("hmm")
print("Welcome")
```

In a nutshell, when the program is run, it keeps asking the user for their password and printing hmm until the user enters test123 – once they do, the program prints one last hmm and then Welcome. Note: The funny symbol on line 2, != means not equal. Thus, this condition says while the response is not equal to the string test123 (...). Let’s go over how while loops work.

If the condition after the keyword while (i.e., response != “test123”) evaluates to true, the program lines inside of the while loop will run (the lines indented under the while, e.g., lines 3 and 4 in this program). But that’s not all!

- Once those indented lines have been executed, instead of moving on to the next unindented line of the program (here line 5), the computer jumps back to the first line of the while loop (here line 2), the one that starts with the word while, and checks the condition again.
- If the condition is true, the computer again executes the indented lines, and again returns to the first line of the while loop to check if the condition is true. This continues to happen until there comes a time when the computer returns to the first line of the while loop and finds that the condition is now false. At that point, instead of repeating the lines inside of the while loop (the indented lines), the computer skips over those lines and jumps to the next line of code that is not part of the while loop (here line 5).

To make this concrete let’s simulate what the computer does when it runs the above program (shown below again to facilitate the code trace):

```python
response = "--"
while response != "test123":
    response = input("Enter password: ")
    print("hmm")
print("Welcome")
```

Can you explain in your own words what happens when this program is “run”?

- **Read line 1**: assign “--” to response
- **Read line 2**: while loop, is the condition true? Yes! response is “--” and that does not equal to “test123”. Move inside the loop
  - **Read line 3**: ask the user to enter a password. Assume user types pass123. assign “pass123” to response
  - **Read line 4**: print hmm
- **Jump to line 2**: is the condition true? Yes! response is “pass123” and that does not equal to “test123”. Move inside the loop
  - **Read line 3**: ask the user to enter a password. Assume user types ABC. assign “ABC” to response
  - **Read line 4**: print hmm
- **Jump to line 2**: is the condition true? Yes! response is “ABC” and that does not equal to “test123”. Move inside the loop
  - **Read line 3**: ask the user to enter a password. Assume user types test123. assign “test123” to response
  - **Read line 4**: print hmm
- **Read line 2**: while loop– is the condition true? No! response is equal to “test123”
- **Jump to line 5**: print Welcome
More on conditions:
The condition after the keyword while tells the computer whether it should execute the lines in the loop (this happens when the condition is true) or whether it should move on to the lines outside of the loop (this happens if the condition is false). Checking conditions allows the computer to adapt to different situations. Determining if a condition is true or false is done by comparing values (note: this holds for variables – the computer compares values of variables, rather than variable names). Examples are below but first you need to know the following facts:

1) == means equal in Python (e.g., 5 == 6 is false because 5 is not equal to 6)
2) != means not equal in Python (e.g., 5 != 6 is true because 5 is not equal to 6)
3) < means less than in Python
4) > means greater than in Python

Suppose we have the following program:

```python
1 counter = 5
2 total = 10
3 name = "mary"
4 print(counter != total)
5 print(name != "ann")
6 print(counter == total)
7 print(name == "ann")
8 print(counter < total)
9 print(counter > total)
```

Extended example: A password checker with a counter

Here is an extension of the above program:

```python
1 response = "--"
2 counter = 0
3 while response != "test123":
    response = input("Password: ")
    counter = counter + 1
4 print("Number of incorrect tries: ")
5 print(counter)
```

Let’s simulate what the computer does when it runs the above program:
• [set program pointer to line 1]  
  • Read line1: assign “--” to response  
     [update model – see Memory]  

• [set program pointer to line 2]  
  • Read line2: assign 0 to counter  
     [update model – see Memory]  

• [set program pointer to line 3]  
  • Read line 3: while loop– is the condition true? Yes!  
    response is “--” and that does not equal to “test123”  
     [update model – see Memory]  

• [set program pointer to line 4]  
  • Read line 4: ask the user to enter a password.  
    Assume user types kazam - assign “kazam” to response  
     [update model – see Memory]
• [set program pointer to line 5]  
• **Read line 5**: increase counter by 1 – counter is now 1  
  [update model – see Memory]

• [set program pointer to line 3]  
• **Jump to line 3**: is the condition true? Yes! **response** is “kazam” and that does not equal to “test123”  
  [update model – see Memory]

• [set program pointer to line 4]  
• **Read line 4**: ask the user to enter a password. Assume user types **pass12**. Assign “pass12” to **response**  
  [update model – see Memory]

• [set program pointer to line 5]  
• **Read line 5**: increase counter by 1 – counter is now 2  
  [update model – see Memory]
• [set program pointer to line 3]
• Jump to line 3: is the condition true? Yes! response is “pass12” and that does not equal to “test123”
  [update model – see Memory]

• [set program pointer to line 4]
• Read line 4: ask the user to enter a password. Assume user types test123. Assign “test123” to response
  [update model – see Memory]

• [set program pointer to line 5]
• Read line 5: increase counter by 1 – counter is now 3
  [update model – see Memory]

• [set program pointer to line 3]
• Read line 3: while loop– is the condition true? No! response is equal to “test123”
  [update model – see Memory]
That’s the end of the lesson!
You have now been introduced to programming using variables, user input, output, and while loops.

Now, how do you think the computer runs a program?
A.4 Pilot Posttest

Question 1
What does the program print?

```
result = 4
result = result + 5
print( result )
```

Question 2
User types “tom” - what does the program print?

```
name = input(“enter your name: ”)
print(“ you entered”, name )
```

Question 3
User types “Z” - what does the program print?

```
char = “Z”
letter = input(“Enter a letter: ”)
res = letter + char
print ( res )
```

Question 4
When answering the questions, the user types “Ontario”, “BC” - what does the program print? Show your work.

```
result = 0
province = input(“enter a province: ”)
result = result + 1

province = input(“enter a province: ”)
result = result + 1

print( result )
print( province )
```
Question 5

What does the program print? Please show your work.

```python
total = 0
while total < 3:
    total = total + 1
    print(total)
print("ha")
print(total)
```

Question 6

The user types “abracadabra”, “kazam” - what does the program print? Please show your work.

```python
secret = "kazam"
answer = "...
while answer != secret:
    answer = input("Guess the word: ")
    print("not quite")
print("he")
```

Question 7

What does the program print? Please show your work.

```python
counter = 0
result = 0
while counter < 2:
    result = result + counter
    counter = counter + 1
print(counter)
print(result)
```

Question 8

The user types “RBC”, “Scotia” - what does the program print? Please show your work.
data = "RBC"
total = 3
while data == "RBC":
    total = total - 1
    data = input("enter bank name: ")
    print(total)
print("You entered this many banks:")
print(total)
Appendix B

Study Materials

B.1 Consent Form
Informed Consent Form

Name and Contact Information of Researchers:

Principal investigator: Kasia Muldner {Kasia.Muldner@carleton.ca}
Veronica Chiarelli {VeronicaChiarelli@cmail.carleton.ca}

Project Title
Learning from Educational Technologies

Project Sponsor and Funder (if any)
NSERC Discovery Grant (Fund 315025)

Carleton University Project Clearance
Clearance #: (116832) Date of Clearance: January 30, 2022

Invitation

You are invited to participate in our research study. This study involves working online on some programming activities with an educational technology. No prior background in programming is required. You are eligible to participate if you do not have any programming experience or very limited experience (no more than one university-level class). The information in this form is intended to help you understand what we are asking of you so that you can decide whether you agree to participate in this study. Your participation in this study is voluntary, and a decision not to participate will not be used against you in any way. As you read this form, and decide whether to participate, please ask all the questions you might have, take whatever time you need, and consult with others as you wish.

What is the purpose of the study?

The purpose of this study is to learn more about how people solve programming problems and what kind of educational technology design best supports that process, including learning from it.

What will I be asked to do?

If you agree to take part in the study, we will ask to complete a brief demographics questionnaire and to solve a set of programming problems. We will record your problem-solving actions in the interface, through the educational technology that will log your entries and/or Zoom (your face will never be recorded). In-session data, such as the chat transcript, will be stored locally on the researcher’s password-protected computer. Operation data, such as meeting and performance data, will be stored and protected by Zoom on servers located in Canada, but may be disclosed via a court order or data breach. We may ask you to explain your answer to one of the problems you solved and will audio record your response (your face will not be recorded). This recording is a condition of participating in the study.

Please note that the recording of problem-solving actions is a mandatory part of the data collection for this study, and so a condition of consenting to participate (as noted above your face will not be recorded). The recording data
will NOT be shared or publicly used nor will it have your name associated with it. The study will take no more than 2 hours to complete (the length of time varies a little by participant).

**Risks and Inconveniences**

We do not anticipate any risks to participating in this study, beyond the fact you may feel some discomfort when you feel you don’t know how to accomplish the programming activities. During the study session, you have the right to not answer any questions and/or end your participation in the study for any reason, by stating that you do not want to continue.

**Possible Benefits**

You may not receive any direct benefit from your participation in this study. However, your participation may allow researchers to better understand how to design educational environments that help students learn how to program.

**Compensation/Incentives**

As a token of appreciation, you will receive $25 or 2% extra credit for CGSC 1001. If you withdraw before finishing the experiment, the amount of compensation will be prorated by time (e.g. if you withdraw halfway through the study, you will receive half the compensation).

**No waiver of your rights**

By signing this form, you are not waiving any rights or releasing the researchers from any liability.

**Withdrawing from the study**

If you withdraw your consent during the course of the study, all information collected from you before your withdrawal will be discarded. Since we do not keep a master list linking your participant ID to your name, withdrawing after the study is not possible.

**Confidentiality**

You will be assigned a participant ID code so that your identity will not be directly associated with the data you have provided. Thus, none of the study materials will have your name on them (we will use anonymous identifiers like P1 instead). However, if you are being granted course credit for taking part in the study, identifying information will be retained by the SONA system using a code until the course credit is granted.

All data, including coded information, will be kept in a password-protected file on a secure computer. Research data will only be accessible by the researchers and the research supervisor and will not have any
identifiable information. We will password protect any research data that we store or transfer. Once the project is completed, research data will be kept and potentially used for other research projects on this same topic.

The results of this study may be published or presented at an academic conference or meeting, but the data will be presented so that it will not be possible to identify any participants.

**Data Retention**

Your Zoom screen recording will be destroyed as soon as it is analyzed (as soon as possible after the study completion and no more than a year). As far as the other data, while the de-identified data will not be shared or made public, we do plan to archive the information for potential future use by Muldner’s research team on related projects and future analysis. Personal identifiers are never attached to the data. Participant contact information will not be kept.

**New information during the study**

In the event that any changes could affect your decision to continue participating in this study, you will be promptly informed.

**Ethics review**

This project was reviewed and cleared by the Carleton University Research Ethics Board B. If you have any ethical concerns with the study, please contact Carleton University Research Ethics Board (by phone at 613-520-2600 ext. 4085 for CUREB B or by email at ethics@carleton.ca. During Covid, the Research Ethics Staff are working from home without access to their Carleton phone extensions. Accordingly, until staff return to campus, please contact them by email.

**Statement of consent – print and sign name**

I voluntarily agree to participate in this study. 

___Yes ___No

________________________  _____________________
Signature of participant      Date

**Research team member who interacted with the participant**

I have explained the study to the participant and answered any and all of their questions. The participant appeared to understand and agree. I provided a copy of the consent form to the participant for their reference.

________________________  _____________________
Signature of researcher      Date
B.2 Demographics Questionnaire

The demographics questionnaire included the following three questions. The first two questions were open-ended and the third question was multiple choice.

What gender do you identify with?

What is your major?

How much prior programming experience do you have?

- None
- High school
- 1 university course
- More than 1 university course

B.3 Pretest

Question 1

What happens when this program is “run”? Refer to each line of the code as you justify your response. For example, line 1 sets the value 1 to the variable result (explain the rest of the program).

```python
1  result = 1
2  result = result + 10
3  print(result)
```

Marks (out of 1):

1 point for 11
Question 2

What happens when this program is “run”? Refer to each line of the code as you justify your response.

```python
1 savings = 100
2 creditcard = 50
3 balance = savings - creditcard
4 print(balance)
```

Marks (out of 1):
1 point for 50
0.5 point for (50) (brackets included but number is correct)

Question 3

What happens when this program is “run”? Refer to each line of the code as you justify your response. When the program prompts the user to enter input, assume that the user types “Donna”.

```python
1 name = input("enter your name")
2 print("you entered")
3 print(name)
```

Marks (out of 1):
1 point for you entered Donna
Or 1 point for
you entered
Donna
Or 1 point for variations of you entered [name that was input]
0.5 for correct with quotation marks
0 if only you entered
0 if only Donna

Question 4

What happens when this program is “run”? Refer to each line of the code as you justify your response. When the program prompts the user to enter input, assume that the user types “Twilight” and then “Harry Potter”.

```python
1  counter = 0
2  book = input("enter a book series: ")
3  counter = counter + 1
4
5  book = input("enter another book series: ")
6  counter = counter + 2
7
8  print("you entered this many book series:")
9  print(counter)
```

Marks:
1 point for you entered this many book series: 3
0.5 point for 3
0 for just you entered this many book series:
-0.5 max for including quotations or parentheses

Question 5

When this program is run, how many times is result from line 3 printed?

```python
1  result = 1
2  while result < 4:
3      print(result)
4      result = result + 1
5  print("goodbye")
```

• I don’t know
Question 6

What happens when this program is “run”? Refer to each line of the code as you justify your response.
Question 7

What happens when this program is “run”? Refer to each line of the code as you justify your response. When the program prompts the user to enter input, assume that the user types “jack” and then “king”.

```python
1 counter = 5
2 variable = 9
3 while counter > 4:
4     variable = variable - 2
5     counter = variable + 1
6 print(counter)
7 print(variable)
```

Marks (out of 2):

2 points (1 point each) for

3

4

−1 If values are printed each time the loop repeats

```python
1 answer = "king"
2 user = "...
3 counter = 0
4 while user != answer:
5     print("try again...")
6     user = input("Guess the card:")
7 counter = counter + 1
8 print("yes!"
9 print(counter)
```

Marks (out of 3):

3 points for the correct answer

*try again...*

*Guess the card: jack*
try again...

Guess the card: king

yes!

1

1 point for each try again...

0.5 for yes!

0.5 for 1

-1 if there is no looping

B.4 Prompts in the Tutor

Prompt 1

What happens when the program below is run?

```
1 name = "Carla"
```

Prompt 2

What happens when the program below is run? Explain what happens at each line of the program.

```
1 response = 0
2 response = response + 1
```

Prompt 3

Explain what happens at each line when this program is run. Assume the user types “yellow” to answer the question.
Prompt 4
How do while loops work? Explain in your own words.

Prompt 5
What makes a while loop stop repeating?

Prompt 6
Which lines in the program below are inside the while loop?

```python
1   number = 0
2   number = 1
3   colour = input("What is your favourite colour?")
4   print("Your choice was")
5   print(colour)
6   print(number)
```

Prompt 7
What happens when this program is run? Explain what happens at each line of the program and what is printed.

```python
1   x = 0
2   while x < 3:
3       x = x + 1
4       print(x)
5       print("Hello")
6       print("Welcome")
```

```python
1   name = "Sheryl"
2   other_name = "Sheryl"
3   print(name == other_name)
4   print(name != other_name)
5   counter = 3
6   num = 1
7   print(counter < num)
8   print(counter > num)
```
Prompt 8

Explain each line of the program below.

```python
1 user = "--"
2 counter = 0
3 while user != "admin":
4     user = input("enter user type:")
5     counter = counter + 1
6 print("Welcome")
7 print(counter)
```

What happens when the program above is run? Assume the user types “guest” then “admin” when prompted to enter a user type.

Prompt 9 Given the program below...

- Why do we include line 1 of the program?

- What happens after line 5 is executed?

- When does this loop stop repeating?

- What is the first line after (or outside of) the while loop?

```python
1 user = "--"
2 counter = 0
3 while user != "admin":
4     user = input("enter user type:")
5     counter = counter + 1
6 print("Welcome")
7 print(counter)
```

B.5 Posttest

The grading scheme for the posttest is omitted as it mirrors that of the pretest.
Question 1
What happens when this program is “run”? Refer to each line of the code as you justify your response. For example, line 1 sets the value 4 to the variable result (explain the rest of the program).

```python
1 result = 4
2 result = result + 5
3 print(result)
```

Question 2
What happens when this program is “run”? Refer to each line of the code as you justify your response.

```python
1 money = 50
2 cost = 10
3 change = money - cost
4 print(change)
```

Question 3
What happens when this program is “run”? Refer to each line of the code as you justify your response. When the program prompts the user to enter input, assume that the user types “Shirley”.

```python
1 name = input("enter your name")
2 print("you entered")
3 print(name)
```

Question 4
What happens when this program is “run”? Refer to each line of the code as you justify your response. When the program prompts the user to enter input, assume that the user types “Toy Story” and then “Lion King”.

```python
1 name = input("enter your name")
2 print("you entered")
3 print(name)
```
Question 5

When this program is run, how many times is result from line 3 printed?

```python
1 counter = 0
2 movie = input("enter a movie title: ")
3 counter = counter + 1
4
5 movie = input("enter another movie title: ")
6 counter = counter + 2
7
8 print("you entered this many movie titles:")
9 print(counter)
```

- I don’t know
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7

```python
1 result = 1
2 while result < 5:
3     print(result)
4     result = result + 1
5 print("goodbye")
```
Question 6

What happens when this program is “run”? Refer to each line of the code as you justify your response.

```
1  counter = 5
2  var = 6
3  while counter > 4:
4      var = var - 3
5      counter = var + 2
6  print(var)
7  print(counter)
```

Question 7

What happens when this program is “run”? Refer to each line of the code as you justify your response. When the program prompts the user to enter input, assume that the user types “king” and then “queen”.

```
1  answer = "queen"
2  user = "...
3  counter = 0
4  while user != answer:
5      print("try again...")
6      user = input("Guess the card:")
7  counter = counter + 1
8  print("woohoo!")
9  print(counter)
```
Note: The correct output values for lines 8 and 9 could be obtained without entering the while loop. For this reason and because not all participant written answers included enough explanation to determine if they traced the code correctly (e.g., with looping) or simply knew how to execute the final three lines of code outside of the while loop, written and oral responses were both considered when assigning a point value.