Informatics in Education, 2021, Vol. 20, No. 4, 583–614

DOI: 10.15388/infedu.2021.26

Understanding Students' Failure to use Functions as a Tool for Abstraction – An Analysis of Questionnaire Responses and Lab Assignments in a CS1 Python Course

Pontus HAGLUND, Filip STRÖMBÄCK, Linda MANNILA
Linköping University, Sweden
e-mail: pontus.haglund@liu.se, filip.stromback@liu.se, linda.mannila@liu.se

Received: February 2021

Abstract. Controlling complexity through the use of abstractions is a critical part of problem solving in programming. Thus, becoming proficient with procedural and data abstraction through the use of user-defined functions is important. Properly using functions for abstraction involves a number of other core concepts, such as parameter passing, scope and references, which are known to be difficult. Therefore, this paper aims to study students' proficiency with these core concepts, and students' ability to apply procedural and data abstraction to solve problems. We collected data from two years of an introductory Python course, both from a questionnaire and from two lab assignments. The data shows that students had difficulties with the core concepts, and a number of issues solving problems with abstraction. We also investigate the impact of using a visualization tool when teaching the core concepts.

Keywords: abstraction, core concepts, procedural abstraction, data abstraction, CS1, Python, functions, prerequisites, parameter passing, scope, references, Python Tutor.

1. Introduction

Controlling complexity is a critical part of problem solving in programming. One important way of doing this is through abstraction. The key to solving a complex problem is not writing a complex solution, but rather to solve several simpler problems. In particular, at the end of a CS1 course a student should have acquired several tools which can be applied to create abstractions when solving complex problems. One of these tools is user-defined functions, which can be used to hide complexity through procedural abstraction and data abstraction. Many of these tools are typically taught in depth, alongside abstraction, in introductory programming courses. Yet many students

struggle with creating and maintaining good abstractions in their code. Perrenet and Kaasenbrood (2006) present students' understanding of abstraction as ranging from the *execution level* to the *problem level*, and Statter and Armoni (2020) found that students need to move between these levels when solving problems. This implies that students who are not proficient with the execution level, or the core concepts, will have difficulties with the higher levels. As such, one possible reason for this struggle is a solid understanding of the tools used for abstraction. For example, it is difficult to utilize procedural abstraction without understanding how functions, parameter passing, references, and return values work. Since Ma *et al.* (2007) pointed out that only 17% of students hold an appropriate mental model of reference assignments at the end of a CS1 course, these issues likely have an impact on students' ability to use abstraction when solving problems.

In this paper we aim to investigate how well students are able to create and maintain abstractions in an introductory programming course in Python, and whether this ability can be improved by highlighting prerequisite concepts such as parameter passing, scope and references using Python Tutor. This is achieved by collecting data from two consecutive years of an introductory Python course in the form of a quiz to assess these prerequisite skills, and solutions to two lab assignments completed during the course. From this data we aim to answer the following research questions:

RQ1 How well do students understand function calls, variable scoping and references at the end of a CS1 course in Python? Does introducing Python Tutor have an effect on students' understanding of these skills?

RQ2 How well are students able to conceive and maintain the abstractions they create during a medium-sized lab assignment?

RQ3 How can we detect, and help students detect, problems with their procedural and data abstraction so that they can improve?

RQ4 Do students voluntarily apply abstraction to other assignments after a lab that heavily scaffolds the use of abstraction?

The remainder of this paper is structured as follows: Section 2 introduces related work, Section 3 describes the course from which data was collected in more detail, Section 4 describes the method used for collecting and analyzing the data, Section 5 presents the results from the data analysis, Section 6 then discusses the results, and Section 7 concludes the paper.

2. Related Work

In this Section we first introduce abstraction in general, followed by a survey of related work on how abstraction is taught and the difficulties in teaching abstraction. Finally, we provide an overview of the importance of fundamental skills, and how these can be taught and assessed.

2.1. *Abstraction*

Due to the broad nature of abstraction as a concept, there are many forms abstraction can take when applied in computing. Liskov and Guttag (2000) highlight three types of abstraction: *procedural abstraction*, *data abstraction* and *iteration abstraction*. Procedural abstraction is the act of using functions or procedures to encapsulate a piece of code that performs a particular and well-defined task, so that other code can use the procedure without having to pay attention to how the task is actually performed. Data abstraction is an extension of this idea to also include data. Thus, a data abstraction defines some kind of abstract data type that is associated with a set of operations that are meaningful to apply to the data. As such, a data abstraction specifies how data is *manipulated* rather than its exact *representation*. Finally, an iteration abstraction provides a way to iterate through a collection of data without detailed knowledge about how the data is stored.

In this paper we will focus on the first two types of abstraction: *procedural abstraction* and *data abstraction*. In particular, we will deal with procedural abstraction through the creation of functions in Python, and data abstraction through the creation of abstract data types (ADTs) as a collection of functions that operate on some data representation (i.e., not using classes in Python).

There are many who argue for the importance of abstraction in computing. As an example, Kramer (2007) likens the ability to create and use abstractions in computing with everyday objects, such as maps and art, to illustrate the importance of being able to remove unnecessary details and focus on what is important. Ginat and Blau (2017) further extends the idea by pointing out that different levels of abstractions exist and are useful at different stages of solving a problem. Due to the importance of abstractions in computing, Moström *et al.* (2008) examined if abstraction should be considered a threshold concept, but found that while abstraction as a whole is likely not a threshold concept, particular types of abstractions (like *procedural abstraction* and *data abstraction*) might be threshold concepts.

Even though abstractions are indeed important in computing, Steimann (2018) points out that some abstractions are counter-productive. What is typically referred to as *leaky abstractions* are examples of such abstraction. In a leaky abstraction, it is necessary for the user of the abstraction to be aware of some aspects of its implementation. The author uses this to point out the importance of learning how certain abstractions are implemented in order to be able to troubleshoot errors related to these leaky abstractions.

2.2. *Teaching Abstraction*

Perrenet and Kaasenbrood (2006) present students' understanding of abstraction in four levels, ranging from what they call the *execution level* to the *problem level*. At the first and lowest level, students understand algorithms as a specific execution, while at the last and highest level they understand them as a black-box that solves a problem regardless of their environment. Statter and Armoni (2020) describe students need to move between these levels when working with algorithms. They further describe that while it is trivial

for an educator to move between these levels (described as wearing different hats), the same is not true for students. Wing (2006) explains that conceptualizing, which is part of computational thinking, requires us to be able to think at multiple levels of abstraction. Hazzan (2008) builds on this saying that awareness of different levels of abstraction is something we should educate our students in. The author reasons that since abstraction is a central theme for computer science, it is important that students are aware of the existence of different abstraction levels and see the benefits of moving between them consciously.

While many agree abstraction is important, teaching it is not trivial. Hazzan (2008) writes that teaching abstraction is not without challenges, which Koppelman and van Dijk (2010) noted is especially true when teaching novices. Hazzan (2008) describes part of the challenge being the lack of rigid rules for abstraction and not being able to teach it in relation to a single topic. It is also not enough to just offer mechanisms for abstraction according to Koppelman and van Dijk (2010). Students should be taught not only how a function works and affects the flow of the program, but also as a tool to hide irrelevant details. Abbott and Sun (2008) finds that teaching the use of existing abstractions is easier than teaching what is needed for students to invent new abstractions. While the authors thinks the recognition of brilliance in abstraction can be taught, teaching students to invent brilliant abstractions is not possible. Likening it to the difference in recognizing brilliance in a script to being able to write a brilliant script, indicating that learning to make something is more difficult than learning to recognize it.

Koppelman and van Dijk (2010) recommend three things for teaching abstraction: start early, teach it consciously and stress the benefits of using it. They do not view abstraction as something that is either mastered or not, but rather like something that is learned gradually over time, which is why you should start early. The authors further write that the teacher should continuously point out where abstraction is used and how that abstraction differs from other abstractions, thus it is important that the instructor teaches it consciously. They also write that students experience abstraction as being complex and not easy to understand, thus it is important for the instructor to stress the benefits and illustrate how it makes things easier. Sooriamurthi (2009) suggests an exercise where students develop a calendar as a good way of introducing students to many important ideas of software development, amongst others abstraction and decomposition. The goal of the exercise being that students learn to decompose larger problems into smaller pieces and define what responsibility each piece needs to have. An approach that should be mentioned in this context is the Model-First approach to teaching abstraction. As stated by Bennedsen and Caspersen (2004), abstraction is one important skill that this approach intends to develop in students.

2.3. *Teaching Fundamentals*

It is difficult to reason about abstractions without a good understanding of the mechanisms provided by a language to create and maintain these abstractions. For example, in order to understand a procedural abstraction, it is necessary to have basic knowledge of

how functions (or procedures) work in the implementation language. In order to properly reason about the abstraction, it is also important to understand parameter passing and return values. For data abstractions it is also important to understand the difference between values and references, and how they behave in the implementation language.

In this way, it is reasonable to talk about these fundamental skills as being prerequisites for abstraction, similarly to how Nelson *et al.* (2020) examined prerequisites to course topics, such as data structures, object oriented programming, and concurrency. The authors found that many questions fail to assess only course topics, but also assess prerequisites to some extent. This is likely also true for abstraction according to the above reasoning. To find the prerequisite skills assessed by a question, Nelson *et al.* proposed a method for coding questions using a codebook of prerequisite skills presented in their paper.

Students' difficulty with learning these fundamental skills have been studied extensively. Qian and Lehman (2017) provides a good overview of these difficulties in their literature review, where they highlight many difficulties in conceptual knowledge and point out the importance of having a good mental model of the underlying computing environment. Goldman *et al.* (2008) identified the fundamental concepts that were both important and difficult according to a group of experts. Examples of these concepts are *parameter scope – use in design*, *issues of scope – local vs. global* and *memory model – references/pointers*, all of which are important when working with abstractions.

Ma *et al.* (2007) studied how familiar students were with some of these fundamental skills at the end of a CS1 course, and found that only 17% of students held a viable mental model of reference assignment. Further research suggests that visualizations can be used to help students gain a better understanding of these concepts (Ma *et al.*, 2009). There are numerous such visualization tools for Python, for example Python Tutor (Guo, 2013), which we will use in this paper, and UUhistle (Sorva and Sirkiä, 2010).

3. Description of the Course

In this paper, we examine the performance of a group of computer science students during their introductory programming course in Python. This course is given during the first half of the students' first semester. Before and during the course, students are given a brief introduction to basic UNIX tools, and towards the end of the course, students build a simple web application as a project. Aside from these two other tasks, the Python course has the students' full attention.

The course focuses on problem solving using imperative programming in Python. No previous programming experience is required, but the majority of students have some previous experience. Lectures introduce general programming structures and techniques while lab assignments aim to develop and test the students' grasp of these techniques and structures. The labs are typically conducted in pairs using pair programming. The students have access to TAs during most days. The TAs can aid students in solving the labs and grasping the techniques. There are also lessons given by the course leader which are optional and specifically targets students without previous programming experience.

Dojos are another optional learning opportunity during the course, and are offered at three occasions. The majority of students choose to attend the dojos but as expected a smaller quantity choose to attend the lessons.

In order to achieve a passing grade or better in the course students need to complete two parts. The first part requires the students to complete the lab assignments. The second part is an exam. The first part consists of 7 assignments, focusing on different concepts:

1. Basic IO and data types.
2. Control structures.
3. Introduction to functions and parameter transfer.
4. Introduction to abstract data types.
5. Abstract data types and procedural decomposition.
6. Higher order functions.
7. Algorithms and files.

The solutions studied in this paper are gathered from labs number 5 and 7. Once a student or pair of students feel that they have solved a lab, a TA will inspect the solution and verbally quiz the students on their solution and their understanding of the concepts. If the solution and answers are adequate, the students are considered to have passed that assignment.

The exam is given in a controlled computer environment at the university. The students are given five problems to choose from and are required to solve at least two for a passing grade. During this exam the students have access to a Python programming book, the official Python documentation, as well as a single double sided page (A4) of notes. The exam is five hours long. Once a student has solved a problem they submit it to course staff. Within 15 minutes they receive a response indicating if the problem is solved in an adequate way. If not, the student is told in what way their solution is inadequate, and has the opportunity to fix their solution and try again. As long as the student makes meaningful strides towards an adequate solution with each attempt there is no limit to the number of attempts, except for the practical limit imposed by time constraints.

4. Method

We collected data from two consecutive years of the course, which we refer to as years 1 and 2. The main difference between the two years is that Python Tutor was introduced in the second year in order to better illustrate functions, parameters, scope, and references to students. The visualization tool was used to illustrate programs during lectures and as part of an assignment. The first demonstration occurred prior to assignment 3 in the course, a few weeks before assignment 5 started. This demonstration centered around why different data types are seemingly treated differently during parameter transfer to functions, specifically why lists passed as parameters to function can have side effects outside the function while some other types of data do not.

Assignment 3 contained a small mandatory task that had to be completed before starting work on the rest of that assignment. During this task the students had to create 4 small programs that replicated a visualization from Python Tutor. The illustrations were: 1) create a list and have two variables reference that same list, 2) create a list and have two variables reference two distinct but identical lists by using copy, 3) calling a function with a list modifying the list outside of the function by use of reference, and finally 4) calling a function with a list as a parameter and using copy to avoid modifying the list outside of the function. Their solutions were then to be presented to and discussed with a TA at their earliest convenience, though they were allowed to proceed working on the rest of the assignment in the interim. Python Tutor was also used to illustrate recursive function calls during a lecture, this demonstration was after assignment 5 but before assignment 7 in the course.

An additional difference is that student interactions were limited during the second year due to the ongoing pandemic. Normally students work in pairs on the same computer, but due to the pandemic they had to collaborate in other ways. The suggested way was to use screen-sharing to let both students in the pair see the contents of the screen without having to be physically close to each other. As the students were still allowed to be physically present in the same room, they could, however, talk to each other as normal anyway.

Two types of data were collected during these two years: answers to a quiz and the solutions to the lab assignments. We examine these in more detail below.

4.1. *The Quiz*

The students were given a quiz as a preparation for the final exam. The quiz consists of 9 questions. Each question presents the student with a piece of code and asks the student to trace the code in order to find the value of one to three variables as indicated. The quiz was implemented as a web page that students access through either their phone or through their laptop. The web page asks for the type and value of each of the variables, and performs basic validation of data input by students. For example, it verifies that lists contain integers and are well-formatted. The questions are then automatically graded when all questions are answered, and students can see their score as well as the correct answer to all questions. The code for all questions on the quiz are shown in Table 1. From the table, we can see that the first four questions mainly covered control structures and appending elements to lists, and that the latter five questions introduce functions and explore scope, parameters, and references as implemented in Python.

Students were given the quiz during the second half of the final two-hour lecture in the course. At that time, students received an e-mail containing the link to the quiz, and were asked to complete the quiz individually during the remainder of the lecture. They were also asked not to execute any of the code while completing the quiz. Paper copies were also available for students who were for some reason unable to complete the online version of the quiz.

In order to arrive at an answer to RQ1 (how well students understand function calls, variable scoping, and references, and if using Python Tutor improves students' understanding), it is necessary to connect students' answer to the questions to the skills assessed by each question. The first step of this process was to find which skills are assessed by each question. As shown by Table 1, most of the questions ask for the value of more than one variable, and in most cases these variables highlight different areas of the concepts involved. For example, in 7 it is possible to realize that the value of b is 10

Table 1

The nine questions in the quiz, including the answers to each question. Each question is followed by the following prompt: "What is the value of the variables after executing the code below?"

Question 1	Question 2	Question 3

`a = [1, 2, 3]`
`b = 0`
`for i in a:`
`  b = b + i`
 	
`# What is b here?`
	`a = [4, 2, 8, 1]`
	`b = 0`
	`for i in a:`
	`  if i > 3:`
	`  b = b + i`
	
	`# What is b here?`
		`a = 0`
		`b = []`
		`while a < 5:`
		`  b.append(a)`
		`  a = a + 1`
		
		`# What are a`
		`# and b here?`

Answer: $b = 6$	Answer: $b = 12$	Answer: $a = 5$, $b = [0, 1, 2, 3, 4]$
Question 4	**Question 5**	**Question 6**

`a = [2, 4]`
`b = []`
`for i in a:`
`  c = 0`
`  d = 0`
`  while c < i:`
`  c = c + 1`
`  d = d + c`
`  b.append(d)`
 	
`# What is b here?`
	`def f1(a, b, c):`
	`  a = a + 5`
	`  b.append(2)`
	`  c = [1, 2]`
	
	`a = 1`
	`b = [1]`
	`c = [1]`
	`f1(a, b, c)`
	
	`# What are a, b`
	`# and c here?`
		`def f1(a, b):`
		`  a['v'] = 4`
		`  b = {'v': 3}`
		
		`d = {'v': 1}`
		`e = {'v': 2}`
		`f1(d, e)`
		`a = d['v']`
		`b = e['v']`
		
		`# What are a`
		`# and b here?`

Answer: $b = [3, 10]$	Answer: $a = 1$, $b = [1, 2]$, $c = [1]$	Answer: $a = 4$, $b = 2$
Question 7	**Question 8**	**Question 9**

`b = 10`
 	
`def f2(a):`
`  b = 20`
`  return a + b`
 	
`c = f2(b)`
 	
`# What are a, b`
`# and c here?`
	`def f3(l):`
	`  l.append(5)`
	`def f4(m):`
	`  m.append(7)`
	`a = []`
	`b = []`
	`c = a`
	`f3(a)`
	`f3(b)`
	`f4(a)`
	
	`# What are a, b`
	`# and c here?`
		`def f5(b):`
		`  b = b + 3`
		`  return b`
		
		`a = 1`
		`b = 2`
		`c = f5(a)`
		
		`# What are a, b`
		`# and c here?`

Answer: a is undefined, $b = 10$, $c = 30$	Answer: $a = [5, 7]$, $b = [5]$, $c = [5, 7]$	Answer: $a = 1$, $b = 2$, $c = 4$

by realizing that the assignment inside `f2` creates a new variable in a separate scope, but fail to compute the value of c due to a lacking understanding of parameter passing for example. Because of this, we treat all questions that involve more than one variable as a multi-part question during the analysis.

Since all skills involved in this quiz are skills that are typically considered to be prerequisites for other courses, we used the method proposed by Nelson *et al.* (2020) to find which skills are assessed by each part of each question in the quiz. First, two researchers independently coded the skills assessed by all parts using the codebook proposed by Nelson *et al.* (2020). Since the codebook did not contain any skills corresponding to dictionaries, we added such a skill when coding the questions. After both researchers independently coded the questions, they met and discussed their coding until an agreement was reached.

These skills can then be used to connect students' skills to their responses on the quiz using a statistical model. In the model, we describe each student in terms of their proficiency $(0 \leq p_i \leq 1)$ with the skills that are assessed by the quiz. Using these skills, we then model the probability of the student answering a particular part correctly as the sum of the skills assessed by that particular part, scaled to the range $[0, 1]$. Assuming that c_j is 1 if the student answered correctly, and 0 if the student answered incorrectly, we can summarize the model as:

$$c_j \sim \text{Bernouli}\left(\sum_{i=1}^{n} p_i s_{ij} \Big/ \sum_{i=1}^{n} s_{ij}\right)$$

Where s is a matrix where $s_{ij} = 1$ if part j in the quiz assesses the skill i, and n is the number of skills. In order to be able to compare skills between the two years studied in this paper, and to get statistically significant results, we modify the model to encompass the skills for an entire year. We do this by assuming that p_{iy} $(y \in \{1, 2\})$ is the average skill for year y, and that c_{jxy} is the correctness of student x in year y's answer to part j. Using these variables, we can write the model for comparing the two years as follows:

$$c_{jxy} \sim \text{Bernouli}\left(\sum_{i=1}^{n} p_{iy} s_{ij} \Big/ \sum_{i=1}^{n} s_{ij}\right)$$

We then approximate this model as a generalized linear model (Nelder and Wedderburn, 1972) with a logistic link function to find the values p_{iy} and test the pairs of hypotheses $H_0 : p_{i1} = p_{i2}$ vs. $H_1 : p_{i1} \neq p_{i2}$ for all i to find differences between the years. We also investigate whether there were any significant difference between success rates on any of the question parts between the two years using a similar logistic model, which models the probability of successfully answering each question as a separate variable.

4.2. *Lab Assignments*

In this Section we present the examined lab assignments, lab 5 and 7, in further detail, as well as what data was collected from the two years.

4.2.1. *Sokoban*

In this assignment, the fifth assignment of the course, students are asked to implement the game Sokoban. In the game, a worker is able to move in a grid with the goal of pushing crates into specified locations. The worker is constrained by walls at certain locations in the grid. The assignment emphasizes the need for abstraction (which is covered in the course prior to the lab) and provides a few requirements that encourage students to think about their abstractions. The lab is presented to the student in four parts in order to scaffold the process. The first part strongly emphasizes creating an ADT to represent the board. The second part encourages them to load a board from a text representation of the board stored in a file (see Fig. 1). The third part is implementing collision detection and the forth is putting together a complete game, including menus, player input and so on. After each part students are encouraged to discuss their thinking with a TA.

In the scope of this course, ADTs are implemented using only functions and built in data types in Python. Classes are not part of the course. As such the students are themselves responsible for not violating the interfaces they create.

Aside from the rules of the game and the format of the input data, the assignment is fairly open-ended, but with guidance available on a daily basis from TAs. Aside from a few requirements, there is a lot of room to solve the lab as you see fit, including how you represent the board. The requirements are as follows:

- Your solution should contain a function `player_can_move` that determines whether the player character (the worker) can make a particular move. Either in a direction or to a given square on the board.
- Your solution should contain a function `crate_can_move` that determines whether a crate can be moved. Either in a direction or to a particular square on the board.

```
############
#..  #     ###
#..  # ○  ○  #
#..  #○####  #
#..    @ ##  #
#..  # #  ○ ##
###### ##○ ○ #
  # ○  ○ ○ ○ #
  #    #     #
  ############
```

Symbol	Meaning
@	Worker
○	Box
#	Wall
.	Storage space
(space)	Floor
⋆	Box on storage space
+	Worker on storage space

Fig. 1. Example of a map (left) as it appears in a text file with sample maps given to students, and the meaning of each symbol (right). Even though not explicitly required, this is also how all submissions output the board during gameplay.

- Your solution should contain a function for displaying the board.
- Your solution should contain a function for loading a board from a text file.
- You are not allowed to store space characters in your representation of the board.

The last requirement is in place to prevent students from storing the board as a 2D-matrix using built-in types. By preventing students from storing any representation of the floor, many manipulation of the data structure that circumvents the interface becomes cumbersome enough to make it easy for students to see that a proper interface is necessary and useful. The creation and use of an ADT to represent the board is heavily emphasized during the lecture leading up to the lab and in interactions with TAs.

From each of the submitted solutions to this lab we extracted the following information:

- What ADTs were present in the code. We identified *board* (the game board), *thing* (common ADT representing worker, box, wall, storage space, and floor), and *level* (one or more boards in a representation suitable to store on a file).
- Information about functions. For each function: how long is it, and which ADTs are it a part of (i.e., which abstractions does it break).
- Are the following aspects of the logic implemented as a free function, or do they belong to a particular ADT?
  - Displaying the menu.
  - Drawing the screen.
  - Processing input.
  - Determining if a move selected by the player is legal.
  - Performing the actual movement.
  - Checking if the game has ended.
- Where does the implementation decide how a specific square should be displayed?

4.2.2. *Copyright*

In this assignment, the seventh assignment of the course, students are asked to implement a small utility that replaces all text between `BEGIN COPYRIGHT` and `END COPYRIGHT` in a source file with the contents of another text file that contains the desired text. The program should either operate on one or more individual files, or on an entire directory. This choice in functionality should be controlled by passing command line arguments to the program. It should also be able to rename files if the user desires to do so, which is also controlled by command line arguments. Aside from specifying the functionality and the command line parameters, the assignment does not specify any particular requirements to guide the implementation in one way or another, but as this assignment is after the Sokoban assignment, students should have the tools to apply abstraction to simplify the problem. Guidance from TAs is still available during most days.

As this problem does not contain any ADTs per se, we focus on abstracting the functionality using functions that can be called multiple times. For each of the submissions,

we examine in which functions each of the following high-level steps are performed, as well as if the functionality was duplicated:

1. Read and process the command line arguments.
2. Find the files to process (if a directory was given).
3. Read the file with the copyright text.
4. Read the file(s) that will be modified.
5. Modify the file contents.
6. Write the file(s) back to disk.
7. Rename the file (if desired).

For each solution, we assigned each function a number based on the order in which they were declared in the source file, and recorded the function (or functions in some cases) where each step was implemented. Any code in the global scope was treated as if it was inside a function with the number zero. Solutions where all code was placed in a single *main* function with no parameters was treated as if it was located in the global scope. In order to be able to compare different solutions, we normalized the recorded numbers by re-labeling functions based on the order in which they appear in the list of high-level steps. For example, one solution might be recorded as `0-2-0-1-1-1-0`, meaning that step 1 is in the global scope, step 2 is in the second declared function, and so on. This becomes `a-b-a-c-c-c-a` after normalization. The letter `a` is used for the first function that appears in the sequence, `b` for the second, and so on. We call this form a solution's *signature*. This preserves data about what functionality was implemented in the same function but allows easy comparison of solutions, since the solutions `0-2-0-1-1-1-0` and `0-3-0-2-2-2-0` both become `a-b-a-c-c-a` after normalization.

For each submission, we also recorded three high-level properties related to abstraction: whether any global variables were used, whether there was a function that modified a single file, and whether or not the file with the copyright text was read more than once if multiple files were processed. We treated any variables that were used to carry data between functions outside of parameters or return values as global variables. Since it is not relevant to talk about global variables in solutions where all code is in the global scope (whether inside a *main* function or not), these were handled as a special case in terms of global variables. Ideally, a student would create one function that is responsible for updating the copyright information in a single file, only read the copyright data once, and does not use global variables.

5. Results

In this section, we present the results of the empirical analysis in this paper. First, we present the results to the quiz, and then the results from analyzing the solutions to the lab assignments.

5.1. *The Quiz*

A total of 59 answers to the quiz were collected across the two years, 34 from year 1 and 25 from year 2. In Fig. 2, we present the amount of correct answers to each part of each question in the quiz. From this overview, we can see that more than 50% of students in both years answered incorrectly on parts 4b, 5a, 5c, 6b and 8c. We can also see that year 2 performed slightly worse overall compared to year 1, but only 3b and 6b were significant on a 95% level. On parts 6a, 7a and 7b students in year 2 did, however, perform slightly better than students in year 1, even though this difference is not statistically significant.

The results from coding each part of the questions in the quiz are shown in Fig. 3. From this, we can see that the quiz assesses 12 skills in total, and that the skills *simple statements*, *assignments*, *tracing* and *operators* are assessed by all parts. Using this information, we can find the combination of skills assessed in the five difficult parts in Fig. 3. For part 4b, we can see that it assesses the same skills as parts 1b and 2b (which also assesses *conditionals*), which means that its structure (nested loops) is likely the reason for this difficulty. Parts 5a and 5c both asses *functions: parameters*, *functions: scoping* and *values and references* in addition to the common skills. Since students performed better on part 5b, where the variable *was* altered in the function, this points towards difficulties in either *functions: parameters* or *values and references*. We see the same situation in part 6b, where the variable is *not* changed by the function, just like in part 6a. Finally, we can see that part 8c is the only part of question 8 that assesses *values and references*, which once again points to issues with this category. This reflects the observations of Ma *et al.* (2007), who also pointed out students' difficulties with references.

5.1.1. *Comparing Skills*

Based on the coding presented above, we model each year's proficiency with these 12 skills using a statistical model as outlined in Section 4.1. Since some skills, namely *simple statements*, *assignments*, *tracing* and *operators*, always appear together they were merged into a single abstract skill as the model would not be able to tell them apart anyway. Fur-

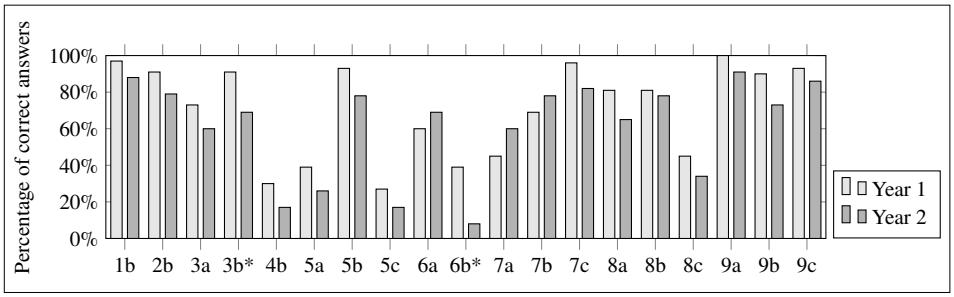


Fig. 2 Overview of the answers from the quiz. Answers marked as "I don't know" are excluded. Parts with a significant difference (on a 95% level) are marked with an asterisk.

Part	array iteration	arrays	assignments	conditionals	dictionaries	functions: parameters	functions: return	functions: return values	functions: scoping	indirection	loop constructs	operators	simple statements	tracing	values and references
1b	✓	✓	✓								✓	✓	✓	✓	
2b	✓	✓	✓	✓							✓	✓	✓	✓	
3a			✓								✓	✓	✓	✓	
3b		✓	✓								✓	✓	✓	✓	
4b	✓	✓	✓								✓	✓	✓	✓	
5a			✓			✓			✓			✓	✓	✓	✓
5b		✓	✓							✓		✓	✓	✓	✓
5c		✓	✓			✓			✓			✓	✓	✓	✓
6a			✓		✓	✓				✓		✓	✓	✓	✓
6b			✓		✓	✓						✓	✓	✓	✓
7a			✓						✓			✓	✓	✓	
7b			✓						✓			✓	✓	✓	
7c			✓			✓		✓				✓	✓	✓	
8a		✓	✓			✓				✓		✓	✓	✓	
8b		✓	✓			✓				✓		✓	✓	✓	
8c		✓	✓			✓				✓		✓	✓	✓	✓
9a			✓			✓			✓			✓	✓	✓	✓
9b			✓						✓			✓	✓	✓	
9c			✓			✓	✓					✓	✓	✓	

Fig. 3. Skills assessed by each part of each question in the quiz.

thermore, since this merged skill appears in all questions, it can also be interpreted as the overall skill level for each year. After fitting the model to the data, we performed 12 t-tests, comparing the performance of each skill between year 1 and year 2. This resulted in 3 skills with significant differences (i.e., $p < 0.05$). Note that due to the nonlinearity of the model, it is not necessarily relevant to quantify the size of the differences.

- The proficiency with the merged skill (*simple statements*, *assignments*, *tracing* and *operators*) decreased significantly from year 1 to year 2 ($p = 0.0438$). This is not surprising, since one interpretation of this skill is the overall proficiency level of the years, and the overall results, presented in Fig. 2, decreased from year 1 to year 2.
- The proficiency with *functions: scoping* increased from year 1 to year 2 ($p = 0.0393$). This is consistent with the increased amount of correct answers for parts 7a and 7b, even if these increases were not significant on their own.
- The proficiency with the skill *indirection* also increased from year 1 to year 2 ($p = 0.0271$). This is not as clearly visible in Fig. 2.

5.2. *Lab Assignments*

In this Section we present the results from the analysis of the lab assignments. To preserve the anonymity of the participants, we have anonymized all code examples presented in this section. This was done by altering some lexical aspects of the source code, for example replacing function-and variable names with synonymous ones, and re-ordering if-statements. Care was taken to preserve the semantics and intent of the original code, in order to not affect the results presented below.

5.2.1. *Sokoban*

In total 36 submissions were analyzed, 21 from year 1 and 15 from year 2. Table 2 shows where the display symbol for each square was determined in the program. We found three possible locations of the logic to determine the display symbol for a particular square. They are as follows:

1. The choice of how the worker looks is not dependent on the chosen representation of a worker in the board ADT.
2. The choice of how the worker looks is somewhat dependent on the chosen representation of the worker in the ADT. The student(s) stores one symbol to represent the worker in the ADT and when displaying the board translates that symbol directly to another.
3. The choice of how the worker looks is dependent on the chosen representation of the worker in the ADT. When displaying a square on the board the students prints the symbol for the worker stored in the ADT.

Most students use option 3, which is the worst option since these submission shows no awareness (or ability to see the benefit of) separating what is displayed and what is stored. The majority of the submissions store symbols that have a direct connection to the look of a square when displayed. 5 out of 36 submissions have no dependency between the internal representation of a square on the board and what is displayed to the player. Of the other submissions the majority stores the symbol that will be displayed directly in the ADT. The submissions from year 2 have a larger part of the submissions directly store the symbol to be displayed in the ADT. Using an option other than 1, creates several problems that students need to deal with in other parts of their solution. In this game it is important that the player can distinguish between a worker that is sitting on a square that contains a storage space and one that is not. The same is true for boxes. When this connection between what is stored and what is displayed exists the students

Table 2

The location of the code responsible for deciding the appearance of each square.

Location	Total		Year 1		Year 2	
1: Independent of board	5	14%	3	14%	2	13%
2: Other symbols in board	8	22%	7	33%	1	7%
3: Same symbols in board	23	64%	11	53%	12	80%

have to constantly check if the symbol stored needs to be altered when moving the worker and/or box, which we can see in Listing 1. These cases are as follows:

1. From a storage square to another storage square.
2. From a storage square to a floor square.
3. From a floor square to a storage square.
4. From a floor square to another floor square.

Fig. 4 shows where students have implemented different functionality in their program. Specifically it shows which ADTs encapsulation is broken by the functionality.

```
def move_worker(board, direction):
    # ...
    if worker_status == 'worker_on_storage':
        if oject_at_to_pos == 'storage_location':
            board[pos_after_move] = 'worker_on_storage'   # 1
        else:
            board[pos_after_move] = 'worker_on_floor'             # 2
        board[pos_before_move] = 'storage_space'
    else:
        if oject_at_to_pos == 'storage_location':
            board[pos_after_move] = 'worker_on_storage'   # 3
            del board[pos_before_move]
        else:
            move_object(board, direction, pos_before_move)       # 4
    # ...
```

Listing 1. Abbreviated example of problems arising from the connection between what is stored and what is displayed. When moving the worker there are 4 different branches of an if-statement that needs to exist to account for different combination. This example fits in the "Other symbols in board" category of Table 2. It also violates the board ADT.

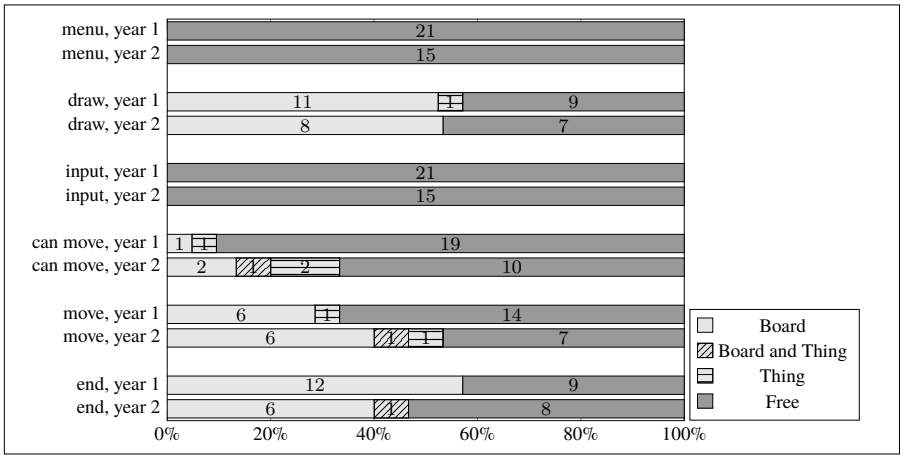


Fig. 4. Overview of which ADT(s) the function implementing each of the six parts of the assignment belong to (i.e., which ADT each of them violate). The x-axis represent percentage of solutions, so that the two years can be compared even though they had different number of solutions. The numbers inside the bars are the number of solutions, not percentages.

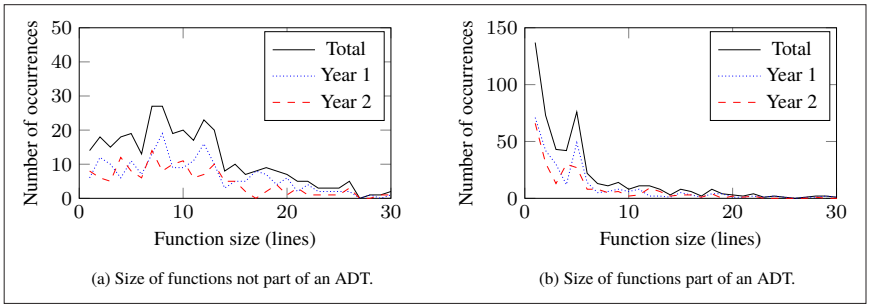
No submissions break the encapsulation of the ADTs when handling the menu and player input. Most submissions from year 1 and year 2 break the encapsulation when drawing the board, with year 1 doing it to a slightly larger extent. Most students do not break the encapsulation when determining if a move is legal or not, though year 2 break the encapsulation to a greater extent. A larger portion of submissions break the encapsulation when moving the character than when determining if the character can move, again year 2 does this to a greater extent than year 1. When figuring out if the game is over, most students in year 2 manage to do this without breaking the encapsulation but not in year 1.

Table 3 shows that the majority of the total submissions and the majority of submissions from each year only used one ADT, which was *board*. A larger part of student submissions from year 2 also introduced an ADT to represent each *thing* on the board. Upon analyzing the solutions only one solution actually treated *thing* as a distinct ADT in relation to *board*. The ADT established for *thing* was mostly either not utilized in the solution, or had its interface violated by *board*. The same solution that treated *thing* as a separate ADT also introduced a third ADT to handle loading of levels.

Fig. 5a shows that a large part of user defined functions in submissions were very short, only one or two lines in length. The length of a function represents the number of lines in the body of the function, excluding empty lines and comments. In both year 1 and 2 functions consisting of only 1 line of code were the most prominent. When compared to Fig. 5b we can see that the majority of all smaller function (6 lines or less) are part of ADTs. In total 879 functions were used by students over all of the submissions. The average length of these functions were 8.7 lines of code and the median

Table 3
Number of solutions that introduce particular ADTs in each year.

ADTs	Total		Year 1		Year 2	
Only Board	20	56%	12	57%	8	53%
Board & Thing	15	42%	8	38%	7	47%
Board & Thing & Level	1	2%	1	5%	0	0%



(a) Size of functions not part of an ADT.

(b) Size of functions part of an ADT.

Fig. 5. Size of functions in the Sokoban assignment.

length was 5.0. Of the 879 functions 524 were part of an ADT. The ADT functions were 6.3 lines in average length and 4.0 in median length. Amongst the 355 functions that were not part of an ADT the average length was 12.4 and the median length was 10.0. All these averages and medians have been rounded to 1 decimal point. Functions that were part of the ADT were in the majority and were shorter on average with fewer outliers of extreme length. Though the longest functions submitted (306 lines) was a part of an ADT.

5.2.2. *Violating an Interface by Accident*

When students violate the ADT's interface they often do so without cause. This is often due to not having created the necessary functions in the ADT, though sometimes the necessary functions actually exist but are not used. As we can see in Listing 1 and Listing 2 the interface is violated since functions of the underlying container are called from a function that should be outside the ADT. This example illustrates both the lack of necessary functions and not using existing functions. Specifically, the submission in Listing 2 contains the needed functions to make the insertions of things in the board but lacks the needed function(s) to remove existing things. The functions created that manages insertion is used in other places in the code.

5.2.3. *Indications of Code Duplication*

The longest submitted function was 306 lines long (counting only the body of the function and not counting empty lines or comments) and originating from year 2 of the submissions. It is a function responsible for moving the worker and aptly named `move_worker`. It requires the direction in which to move the worker, represented by a character (`w`, `a`, `s`, or `d`) for up, left, down, and right. It also requires a board which is an abstract data type. The extreme length of the function occurs because of code duplication.

Only two lines, of which one is an empty return statement on the last line, out of 306 are executed on all code paths, regardless of the direction of the movement as we can see in Listing 3. The remaining 300 lines (discounting the if-statement itself) of the function is really 75 lines repeated 4 times, once for each direction with magic numbers altered , as we can see in Listing 4.

```
def move_thing(board, xcord, ycord, direction):
    thing = get_object(board, xcord, ycord)
    # ...
        character = next_thing(board, xcord, ycord, direction)
        if thing == '+':
            board.append([xcord, ycord, '.'])
        if character == '+':
            board.remove([xcord + 1, ycord, '.'])
        board.remove([xcord, ycord, thing])
        board.append([xcord + 1, ycord, character])
    # ...
```

Listing 2. Example of needless ADT violations. Functions exist for handling the insertions but are not used, and functions for deletion could easily be added.

```
def move_worker(board, direction):
    xcord, zcord, ycord = locate_player(board)
    if direction == 'w':
        # move character up
    elif direction == 'a':
        # move character left
    elif direction == 'd':
        # move character right
    elif direction == 's':
        # move character down
    return
```

Listing 3. Abbreviated example of a submitted function to move the worker. The unabbreviated version contains 306 lines of code.

```
def move_worker(board, direction):
    # ...
    if square == '.' and box == 'o' and worker == '@':
        board[xcord-2].remove((ycord, '.'))
        board[xcord-1].remove((ycord, 'o'))
        move(board, 'storage box', xcord-2, ycord)
        coord = board[xcord].pop(zcord)
        board[xcord-1].append(coord)
    # ...
```

Listing 4. Example of one of the branches on an if-statement inside the function in Listing 3 designed to move the worker. It illustrates the operators and magic numbers used for one of the directions.

This way of solving the problem is found in multiple levels of the solution to moving the worker. In Listing 4, we can see 1 of the 8 branches of an if-statement that moves a worker if the worker has to push a box. One branch exists for each combination of circumstances that can exist and another 4 branches exists for moving the worker if there is no box involved. This does not include the checks done to determine if the move is legal at all, which further adds complexity. These 12 cases (and several more that are trivial in nature) are repeated for each direction with slight differences in the numbers or operators used in the function calls. During execution of this function there are 64 different paths that can be taken. While this function is an outlier amongst the submissions in size for a single function, similar ways of solving the problem of moving the worker in 4 directions are not uncommon.

As we can see in Listing 5 another solution instead created 4 functions, one for moving the worker in each direction. The difference between each of these functions is again only magic numbers used to move the worker accordingly. This solution uses 59 lines per function to handle moving the worker, a total of 236 lines. In this example 13 branches of if-statements exists for each direction, totaling 52 branches to move the player in all 4 directions. As in the previous example, each possible scenario that can occur for each direction is presented as a separate branch. One can also note that both Listing 3 and Listing 4 are violating the ADT.

```
def move_north(board):
    # ...
    if not player_can_move(board, x, y-1):
        # ...
    elif get_object(board, x, y-1) == '.':
        # ...
    elif get_object(board, x, y-1) == 'o':
        # ...
    # ...
```

Listing 5. Example of a function specialized to move the worker up. Note the use of operators and magic numbers that is again present.

Looking at all submitted functions in descending order of length, every function that was longer than 32 lines of code had obvious issues with at least procedural abstraction. The functions either operated on too many levels of abstraction, had code duplication or had more than one responsibility. The longest function that had no obvious issues with abstraction, which was 32 lines long, was a function dedicated to loading the game board. This involves doing something special for each symbol in the text-file, which results in longer functions with many cases. Its hard to avoid having one branch for each symbol with the tools available to the students at this stage. The longest function that did not load the board and had no obvious errors was 24 lines long.

5.2.4. *ADT Interface*

It was previously mentioned that the functions containing only 1 line were the most common among the submissions. These functions are part of the ADTs and are used to perform actions for which the underlying containers happen to have matching functionality. An example is adding something to the list in the board ADT, as shown in Listing 6.

These ADT-functions serve to preserve the interface of the ADT and in this particular example whenever a crate needs to be added to the board it is done through the use of this function. This also extends to many of the functions submitted that contained 2 lines, as shown in Listing 7.

Many of the short functions submitted that are part of the ADT are similar to these, often just calling a single member function of the underlying container in the ADT. The existence of functions such as these do not, however, mean that students use them. It is

```
def new_crate(board, x, y):
    board.append([y, x, 'o'])
```

Listing 6. Example of a one line function serving as a simple interface to the ADT.

```
def make_board():
    b = []
    return b
```

Listing 7. Example of a two line function serving as part of the ADT interface.

fairly common that students in spite of creating a fairly robust ADT interface simply circumvent it, and use built-in functionality of the underlying container types instead.

5.2.5. *Copyright*

In total, we analyzed the solutions from 38 student pairs to the copyright problem. Out of these, 22 were from year 1 and 15 from year 2. As mentioned in Section 4.2.2, we examined two aspects of the solutions: where each of the seven high-level steps were located, and the three high-level properties.

Fig. 6 contains an overview of the number of functions that were used to implement the seven high-level steps presented in Section 4.2.2. In a few cases we found small utility functions that did not fit into any of the seven steps (e.g., basic string manipulation). As these functions did not fit into any of the categories, they were not recorded and are therefore not included in Fig. 6. From this overview, we can observe two trends. First and foremost, around half of the solutions (54% in year 1 and 60% in year 2) used at most one function to solve the problem, even though several previous assignments focused heavily on abstraction. Secondly, more solutions from year 2 did not use any functions at all compared to year 1. Regarding duplication, we found four solutions with duplicated functionality in total. One in year 1 and three in year 2.

By examining the solution signatures (i.e., how functionality was distributed between functions), we found that 48% of solutions (or 18 solutions) only appeared once. Out of the remaining 52% of solutions (19 solutions), the most common one (27% or 10 solutions) was to implement all seven steps in the global scope, not using functions. The second most common solution (10% or 4 solutions) was to implement reading and processing of the command line arguments in the global scope and call a single function that performed the remainder of the work. The remaining solutions (15% or 6 solutions) appeared twice. One of these was to extract the last three steps into an additional function, another was to also locate the logic for finding the relevant files in the global scope alongside handling of command line arguments, and the last one was to split handling of command line arguments between code in the global scope and inside the function that does all the work.

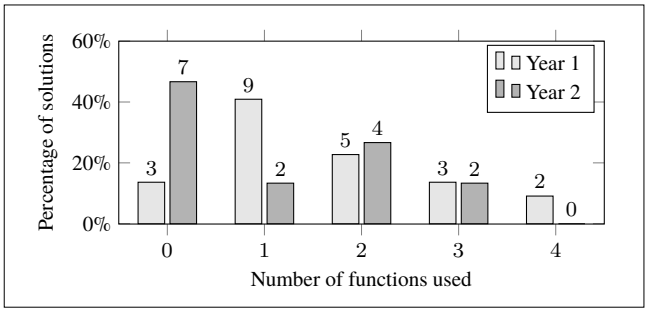


Fig. 6. Total number of functions used to implement the 7 high-level steps of the assignment. The height of each bar is determined by the percentage of solutions that year, so that the two years may be compared even though they had different number of solutions. Numbers above bars indicate number of solutions.

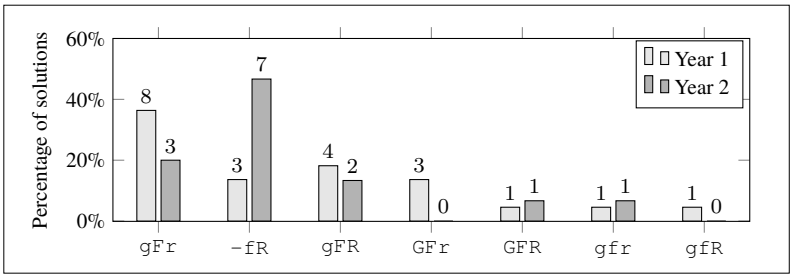


Fig. 7. High-level properties measured. In the labels, we use G = global variables (or -if no functions were used), F = function for processing a single file, R = reading the template file only once. Uppercase letters mean that property was observed, lowercase mean that property was not observed. Height of bars are relative to the percentage of solutions in that year, so that the two years may be compared even though they had different number of solutions. Numbers above the bars are the number of solutions.

Finally, by examining the three high-level properties for the solutions, we can gain some additional insights into the solutions. These are presented in Fig. 7. From there, we can see that 16 out of 37 solutions (43%) read the template file multiple times (all labels with a lowercase `r`). However, this seems to be an unintended side-effect of using functions for abstraction, as all solutions not using functions (label `-fR`) only read the template file once. We can also see that the ideal solution, `gFR` (no globals, a function for processing a single file, and only reading the template file once) was less common than reading the file multiple times (label `gFr`), and not using functions at all (label `-fR`).

Aside from the difference in the two years we have seen previously, that students in year 2 did not use any functions at all to a larger extent than year 1, there are only two major differences between the two years. First, the number of solutions labeled `gFr` (a good solution, but reading the template multiple times) decreased, as did `GFr` (as previously, but using global variables). It is difficult to determine if these difference were due to more students opting to not use functions for abstraction in their solutions, or if this was due to a better awareness of these issues in the second year.

6. Discussion

In this Section we will discuss the results presented in the previous section. We start by discussing the validity and implications of the results from the quiz, which gives an idea of the students' knowledge of the prerequisites for abstraction. After that, we will discuss how the results from the solutions to the two lab assignments can be interpreted and applied in the future. Finally, we will present possible extensions of our analysis that we leave as future work.

An overarching trend from the results is that students in year 1 performed slightly better than students in year 2 in cases where a difference was visible (e.g., on the quiz). This difference could mean that introducing Python Tutor, as previously described, de-

creased students' skills. Since the quiz showed a significant improvement for some of the skills, we believe that these differences are either due to 1) natural variation in prior education and experience in the students between the years, or 2) the reduction of student interactions due to the ongoing pandemic. Typically, students spend most of their day in a computer lab dedicated to this group of students, and they are thus always close to peers with which they can discuss any issues that arise even when TAs are not available. Based on our prior experience with this program, and as noted by Laal and Ghodsi (2012) among others, we believe that this interaction is very valuable for students learning. Therefore, reducing these valuable interactions may very well have an impact on students' learning.

It should also be noted that the data was collected from one student group at one university. As such any generalizations based upon these results needs to be considered in that context and may not be applicable to other student groups, universities, or countries.

6.1. *The Quiz*

Even though each question in the quiz was designed to assess only a small number of skills that were previously identified as problematic, most of them failed to isolate individual skills as shown by the coding of the questions in Fig. 3 in Section 5.1. Even though the answers indicated that *values and references* and *functions: parameters* are difficult concepts, it is difficult to reliably assess all skills in isolation. For example, to assess whether the student knows *values and references*, it is necessary to manipulate some data, which in turn involves other skills. It is possible to address this problem by assessing the more basic skills individually in other questions. Doing this for all skills increases the length of the quiz, which in turn increases the time and effort required by students to complete the quiz. Furthermore, the initial questions may be experienced as too easy and therefore not relevant to students, which may decrease students' motivation to produce the correct answer and continue with the quiz.

It is, however, possible to avoid these problems by ensuring that the skills assessed in the questions do not fully overlap and then compute the proficiencies from the results by using a statistical model as presented in Section 4.1. This way, questions can be kept at a similar difficulty level, and the number of questions can be kept low, while retaining the ability to observe students' proficiency with individual skills. In this paper, we use the model to find the overall proficiency levels for two years to see whether introducing a lab using Python Tutor improved students' understanding of skills related to functions. Even though the students' overall score on the quiz decreased from year 1 to year 2, the model showed an increase in *indirection* and *scoping*, which are two of the skills the assignment with Python Tutor was intended to cover. Based on this observation, we believe this is a viable approach, at least when applied on groups of students.

There are, however, some considerations that need to be addressed. First and foremost, the model assumes that all skills assessed by one part are equally important to whether or not a student answers correctly. This is likely true for some of the questions.

For example, it is likely equally important to understand *values and references* and *functions: parameters* to answer 8c correctly. In other cases, especially when comparing questions to each other, it is likely that the same skill has a different impact for the outcome of different questions. This could be addressed by allowing other numbers than 1 and 0 when describing which skills are assessed by a question. As these are typically not known beforehand, this means that such a model would no longer be linear, and would contain more parameters, meaning that more data would be needed to find these parameters. Another approach would be to divide skills in the codebook into sub-skills that highlight different areas of the same concept. Then it is possible for the model to assign different proficiencies for declaring an array and reading from an array, for example.

Even though we used the model to examine the proficiencies of two different years in this paper, it is possible to apply the model to individual students to provide individual and automated formative feedback to students taking the quiz. In this case, a model could be fitted to a single student, and the lowest proficiencies could be suggested as areas for further self-study. If used in such a system, the fact that the model assumes that each skill has an equal impact for the outcome will likely become more apparent. Thus, questions would have to be designed so that the involved skills are indeed roughly equally important, or a model with more fine-tuned weights would be needed.

6.2. *Sokoban*

The overarching goal of the Sokoban lab is to give students the opportunity to work with abstraction in a larger project. In part through working with data abstraction and in part through working with procedural abstraction as described by Moström *et al.* (2008). While both these concepts are introduced and practiced in two previous labs, it is at a much smaller scale, and with more robust descriptions. In a really good solution a student manages to create an ADT for one or more parts of the lab, manages to delimit the responsibilities of that ADT, creates an adequate interface, and respects that interface throughout the solution. While some students manage to accomplish this goal, others do not, and there are different indications of this in the results as we will discuss below.

Based on the data from the Sokoban lab, we can see that only a small portion of students arrive at a desirable way of deciding were the look of each square of the board should be computed. The most desirable way of doing it is having no dependence between the representation used in the board ADT and how it is displayed. The least desirable way is storing the actual symbol that is going to be displayed in the ADT, making the look of the board when displayed entirely dependent on the internal representation of the ADT. This indicates a failure to separate the internal representation of the ADT and what is displayed. Instead of viewing the board as a container storing, for example walls at certain coordinates, the majority of students view the board as a list (or dictionary) storing characters at coordinates. The display of the board and the internal representation have no need to be connected to one another. Only 5 out of 36 submis-

sions use the most desirable solution and 23 out of 36 use the least desirable one. We interpret this as the majority of the students not realizing that these should not be or do not need to be dependent on one another. Alternatively, if the students realize this, they fail to see the value of not having them be dependent on one another. Since the first part of the lab is creating an ADT to represent the game board, it is reasonable to conclude that most students are unable to come up with an abstract representation of the board that does not include all these symbols. The students probably reason that they need to store the symbols that will be printed at a later stage somewhere. Therefore, they fail to remove the unnecessary parts of the abstraction, which is presented as an important part of creating good abstractions by Kramer (2007).

Even though the students are allowed to introduce more ADTs in their solution, very few students do so. A few students do introduce a separate ADT to represent individual elements on the board (walls, boxes, etc.) which we call thing. However, in all cases except one where thing is introduced, the abstraction is violated in at least the board ADT. There was also one solution that included an ADT to aid with the loading of levels and that ADT was also used correctly. The introduction and use of a board ADT is heavily emphasized in both the description of the task and in interactions with course staff, which explains why it is present in all solutions. The lack of distinction between thing and board may also indicate that it is harder to differentiate between two ADTs than between what is and is not part of an ADT at all. This can be viewed through the lens of different levels of abstraction, as described by Perrenet and Kaasenbrood (2006), where understanding the difference between one ADT and another requires the student to move between different abstraction levels (i.e., wearing different hats). In other, words the student needs to be able to delimit what is part of a specific ADT, and what is not. They also need to respect that delimitation. The challenge of not violating the interface of an ADT is however not trivial even if there is only a single ADT as discussed below. Students are proficient in handling existing abstractions in Python and all solutions include some kind of container that exists in the Python standard library. While interactions with these containers is handled through the interface provided, it seems challenging for students to develop and use their own interface for their own ADTs. This could be an example of how it is easier to recognize and use abstractions than it is to create ones own, as written about by Abbott and Sun (2008).

When looking at the functions inside and outside of the ADT, we can see that all students manage to separate the handling of menu and player input from the ADT. This is to be expected since they do not really have any need to interact with the functionality of the ADT. Most students do break the encapsulation when drawing the board, indicating that drawing the board is considered a part of the board ADT or that it is a convenient way of solving the issue. The latter is also supported by the way students represent the look of each square on the board. Most students store the actual symbol to be displayed in their ADT. It seems to be hard for students to understand when a function violates an ADT and when it does not, which could indicate a problem in moving between different levels of abstraction, as laid out by Statter and Armoni (2020); Perrenet and Kaasenbrood (2006). It is also worth noting, that drawing the board is

introduced earlier as a problem to be solved in the lab than any of the game rules. Many introduce the functionality of displaying the board at the same time as they are creating the board ADT to see if their ADT is functioning correctly.

While the argument can be made that drawing the board is functionality that should be part of the ADT, determining if the worker can make a certain move, performing a move and determining if the game is over are all components of the program that should be solved without violating the ADT. As the results show, it is however common for students to violate the ADT to some degree when moving the character and determining when the game is over. It is unlikely that a solution violates the ADT when determining if a certain move is legal. The reason for this may be that a fairly strict description of this functionality is provided in the lab. This delimitation of functionality may allow students to focus on the higher levels of abstraction when writing the function. It is also worthwhile noting, that determining the legality of a move and if the game is over only requires reading from the ADT, while the component responsible for moving the worker also has to make changes to the ADT. It may also be easier to make the mistake of conceptually confusing the movement of the worker with moving an arbitrary piece on the game board from one square to another. The latter being a natural part of the ADT while the former is not.

Functions that are outliers in length are problematic and shows a lack of understanding when working with abstraction. As we can see in Listing 5, the longest functions found amongst the submissions duplicate a lot of code. This duplication is indicative of a few things: 1) poor delimitations of the responsibility of the function, leading to a function that does too many things, 2) instead of finding the general case each possible circumstance is explored individually, which leads to enumeration of all cases, and 3) not breaking the problem down into small enough parts, which stems from not finding a more detailed abstraction. Each of the very long functions have poor readability and a large number of paths the program can take. As we can see, not applying proper abstraction to solving the problem leads to a complex solution to a complex problem.

Simple pseudo code for solving the movement of the worker could be: 1) determine legality of move, 2) remove the worker from its current square, and finally 3) insert the worker at the new square. Even then, the question of whether the move is legal should probably be answered before the move function is called. In other words, students that write very long functions are not able to create good abstractions when solving the lab. Even if they follow some of the rules of abstraction, such as creating a function that has a clear purpose, in this case moving the worker in a given direction.

It is interesting that in all analyzed solutions, functions that were longer than 32 lines all had obvious problems with abstraction; mixed abstraction levels, code duplication, ADT violations or problems delimiting the responsibility of the function, or a mix of them all. With that being said, this should not be interpreted as if functions shorter than that were all good. A finer granularity of these estimates could probably be arrived at if one analyzed different parts of the solution individually. Anecdotally, it would seem like functions solving certain problems can be bigger than others without having obvious abstraction issues.

6.3. *Copyright*

Based on the analysis of the solutions to the copyright lab, we can see that many students did not use functions to create abstractions when implementing their solutions. We saw that 57% of solutions (21 of 37 solutions) used zero or one functions when implementing their solution. About half of these implemented their solutions entirely in the global scope, indicating that they did not feel the need for any further abstractions. The remainder of these solutions did separate the handling of command line arguments from the logic of the program, which is definitely a step in the right direction, but did not feel the need to create any further abstractions in the remaining logic. This lack of abstraction was quite apparent even though this lab assignment was done within a couple of weeks of the Sokoban assignment, which heavily emphasized procedural abstraction. This likely means that students did not see the effort of creating abstractions to be worthwhile in this assignment, either because they did not see any major benefits from the previous assignment, because they felt this assignment was simple enough, or a combination of the two.

Another interesting observation is that many of the solutions that introduced a function to process individual files did read the template file multiple times, while solutions that did not only read the file once. This observation highlights some of the dangers of abstractions, as pointed out by Steimann (2018). In this case, students likely failed to see that the comparatively expensive operation of reading the template file only needed to be done once when introducing the abstraction.

The two categorizations of solutions used to analyze this assignment provides interesting insights into the solutions. In this case, the analysis of the three high-level properties provided information that was easier to interpret than the normalized signatures of where the work was performed. This is likely due to the fact that the signature shows more details of the solutions. Therefore, it is necessary to have a larger data set than we had access to in order to unveil more detailed trends in solutions other than the two most common ones. Otherwise, the only thing we can tell from the signatures is what kinds of solutions are present, not how common they are. This is why the less detailed analysis proved more useful. Since it had fewer degrees of freedom, it is better at illustrating some overarching trends, even without large quantities of data.

One shortcoming of the analysis that is worth pointing out is that any small functions that only implemented a small part of the final solution were not included in the signatures, and thereby not in our measure of the number of functions used. Having these kind of functions show that students have realized that some operation was common or difficult to implement and thus abstracted the functionality using a function. This was, however, fairly uncommon in the solutions in this data set, so it does not have a major impact on the data presented in Section 5.2.5. One could also argue that this type of abstraction is different from that of examining the problem to be solved, breaking it into high-level pieces, and structuring the program accordingly. One could also argue that not having functions does not imply that no effort was made regarding abstractions. Since we did not see much duplicated functionality in the solutions,

students have likely identified at least some of the high-level steps and found ways to write code that is general enough to handle all cases, even though this was not put in individual functions.

6.4. *Future Work*

In this paper, the statistical model used to map quiz answers to skills was only used to estimate the proficiency of an entire year. It would, however, be possible to use the same model on a single student to estimate that student's skills, and thus give automated feedback regarding what topics that student might benefit from reviewing. In this case, it is likely important to take into account the different difficulties of the questions, and the fact that some questions require a deeper understanding of a concept than others. Exploring this would, however, require a larger data set, and potentially also interviews to validate the model on an individual level, or connecting the performance on the quiz to performance in the course overall, or indeed performance in future courses. It might also be necessary to revise some of the questions to better pinpoint particular skills.

One interesting observation from analyzing solutions of the Sokoban assignment was that functions with more than 32 lines of code all had obvious abstraction issues. Since this is easy for automated tools to measure, it can be used to give automatic feedback to students during the labs. Cursory findings indicate that this length varies based on the problem solved by the function (e.g., reading the map from a file, moving the worker, etc.). As such, it would be interesting to examine this relation in closer detail for a wider variety of problems. It would also be valuable to explore the possibility of giving automatic feedback regarding violations of an ADT, since many students seem to struggle with realizing these violations. This is, however, more difficult to do automatically, especially in a dynamically typed language like Python.

Even though it might be difficult to incorporate all findings of this paper into tools for automated feedback, these findings are still useful. While the analysis done in this paper is not feasible to perform on all submitted solutions in a course, the analysis reveal where problems occur so that future TAs can be instructed to spot potential problems earlier, and give students better feedback earlier. This reduces both frustration from students, as they do not have to re-do large parts of the assignment, and the time needed for feedback once TAs review the students' code, as it is more likely to be correct and well-abstracted. These insights might also be used to develop targeted code-review tasks where students review each other's code. A similar analysis of other, related, problems may thus be performed for other, similar, problems in order to find how similar issues surface there.

Finally, since the quiz was submitted anonymously, we were not able to correlate individual students' skills to their ability to work with abstractions in the lab assignments. Furthermore, since students' performance with the lab assignments were similar between the two years, we were unable to draw any definitive conclusions in this regard. A follow-up study where it is possible to follow individual students would probably provide deeper insights in this matter, and would therefore be an interesting next step to this paper.

7. Conclusion

In this paper we examined students' ability to create and maintain abstractions in the lab assignments of an introductory Python course. We also examined if introducing an assignment involving visualizing parameter passing, scope, and references in Python Tutor helped students' understanding of these concepts, and had any impact on their ability to work with abstractions.

We collected solutions from two of the seven lab assignments in two years of the course, and found that most students do not create more ADTs than the minimum required by the assignments. We also found that amongst the minority that does introduce more ADTs, almost all have problems with differentiating between multiple ADTs. In this case, students who introduced an additional ADT for a thing inside the board seldom managed to keep the two separate, and often did not use the created interface when reading or writing to things inside the board.

We also found that students have difficulties with differentiating between either 1) functions that are a part of the ADT (and may thus manipulate the internal representation) from those not a part of the ADT (and may thus only use the interface), or 2) that they have difficulties understanding the difference between manipulating the internal representation and using the interface. We also found that all functions longer than 32 lines had some kind of abstraction issues. Finally, we found that many students did not voluntarily introduce their own abstractions in a later assignment which did not explicitly focus on abstraction.

We studied the effect of introducing Python Tutor in two ways. First through a quiz that contained questions about parameter passing, scope, and references. Even though the overall results were lower in the second year, a statistical analysis showed a significant increase in students' proficiency with scoping and references. We also examined differences in the lab assignments between the two years, but did not find any big differences, perhaps since the second year was weaker overall. Therefore, we can not conclude that using Python Tutor on its own has any noticeable affect on students' ability to work with abstractions, but since it seems to improve some prerequisite skills it may very well be one important part of many others that help students improve their abstraction skills.

To conclude, we answer the research questions posed in the introduction:

RQ1 Overall, students understand function calls, variable scoping and references well with the exception of parameter passing. Using Python Tutor to visualize these concept did provide a significant improvement in students understanding of scope and references, even though the second year was weaker overall. The difference was not, however, extreme which implies that the intervention is not a silver bullet.

RQ2 Some students are able to create and maintain the abstractions they create, but most fail to do so. We also found that a majority of the students do not introduce more data abstractions than those required by the assignment, and those that do rarely manage to keep the multiple abstractions separate. The

reluctance to introduce new abstractions was also seen in many solutions with code duplication. Adding some procedural abstraction to these solutions (in particular parameterization and generalization) would have greatly reduced the total amount of code. Finally, we found that certain early choices about the abstraction makes it more difficult to maintain the abstractions later (e.g., whether the look of a square is decided independent of the board or not).

RQ3 Our analysis indicates that function length can serve as an indicator of functions that break abstractions. This could for example be used by students as an indication of when to seek guidance from a TA. Course staff could also use long functions as a starting point for finding abstraction-related issues. The analysis also suggests that the functions for moving the worker and checking if the game is over often violate ADTs without needing to do so, and are thus also good starting points for course staff. Finally, we found that only 5 out of 36 students determine the look of a square independent of the data stored in the board ADT, which makes the implementation more cumbersome. This could be addressed in the assignment, or frequently by TAs, to spare the students much trouble. All of these issues can also be used by teaching staff at the end of a lab to highlight alternative solutions that makes solving the problem easier.

RQ4 Students do not in general voluntarily apply abstraction to a later assignment in the course (the Copyright assignment), even after an assignment that heavily emphasized abstraction (the Sokoban assignment). This is similar to what we saw in the Sokoban assignment, where students rarely introduced more abstractions than what was specified in the assignment, even though it would benefit the students. Better highlighting the benefits of abstraction in a previous lab, as suggested above, might improve the situation.

References

Abbott, R., Sun, C. (2008). Abstraction Abstracted. In: *Proceedings of the 2nd International Workshop on The Role of Abstraction in Software Engineering*. ROA '08. Association for Computing Machinery, New York, NY, USA, pp. 23–30. 9781605580289. https://doi.org/10.1145/1370164.1370171

Bennedsen, J., Caspersen, M.E. (2004). Programming in Context: A Model-First Approach to CS1. *SIGCSE Bull.*, 36(1), 477–481. https://doi.org/10.1145/1028174.971461

Ginat, D., Blau, Y. (2017). Multiple Levels of Abstraction in Algorithmic Problem Solving. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Association for Computing Machinery, New York, NY, USA, pp. 237–242. 9781450346986. https://doi.org/10.1145/3017680.3017801

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M.C., Zilles, C. (2008). Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process. In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '08. ACM, New York, NY, USA, pp. 256–260. 978-1-59593-799-5. https://doi.org/10.1145/1352135.1352226

Guo, P.J. (2013). Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Association for Computing Machinery, New York, NY, USA, pp. 579–584. 9781450318686. https://doi.org/10.1145/2445196.2445368

Hazzan, O. (2008). Reflections on Teaching Abstraction and Other Soft Ideas. *SIGCSE Bull.*, 40(2), 40–43. https://doi.org/10.1145/1383602.1383631

Koppelman, H., van Dijk, B. (2010). Teaching Abstraction in Introductory Courses. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '10. Association for Computing Machinery, New York, NY, USA, pp. 174–178. 9781605588209. https://doi.org/10.1145/1822090.1822140

Kramer, J. (2007). Is Abstraction the Key to Computing? *Commun. ACM*, 50(4), 36–42. https://doi.org/10.1145/1232743.1232745

Laal, M., Ghodsi, S.M. (2012). Benefits of collaborative learning. *Procedia -Social and Behavioral Sciences*, 31, 486–490. https://doi.org/10.1016/j.sbspro.2011.12.091

Liskov, B., Guttag, J. (2000). *Program Development in JAVA: abstraction, specification, and object-oriented design*. Addison-Wesley, Boston, USA.

Ma, L., Ferguson, J., Roper, M., Wood, M. (2007). Investigating the Viability of Mental Models Held by Novice Programmers. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '07. ACM, New York, NY, USA, pp. 499–503. 1-59593-361-1. https://doi.org/10.1145/ 1227310.1227481

Ma, L., Ferguson, J., Roper, M., Ross, I., Wood, M. (2009). Improving the Mental Models Held by Novice Programmers Using Cognitive Conflict and Jeliot Visualisations. In: *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '09. ACM, New York, NY, USA, pp. 166–170. 978-1-60558-381-5. https://doi.org/10.1145/1562877. 1562931

Moström, J.E., Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., Zander, C. (2008). Concrete Examples of Abstraction as Manifested in Students' Transformative Experiences. In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. Association for Computing Machinery, New York, NY, USA, pp. 125–136. 9781605582160. https://doi.org/10.1145/1404520.1404533

Nelder, J.A., Wedderburn, R.W.M. (1972). Generalized Linear Models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3), 370–384. https://doi.org/10.2307/2344614

Nelson, G.L., Strömbäck, F., Korhonen, A., Begum, M., Blamey, B., Jin, K.H., Lonati, V., MacKellar, B., Monga, M. (2020). Differentiated Assessments for Advanced Courses That Reveal Issues with Prerequisite Skills: A Design Investigation. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '20. Association for Computing Machinery, New York, NY, USA, pp. 75–129. 9781450382939. https://doi.org/10.1145/3437800.3439204

Perrenet, J., Kaasenbrood, E. (2006). Levels of Abstraction in Students' Understanding of the Concept of Algorithm: The Qualitative Perspective. *SIGCSE Bull.*, 38(3), 270–274. https://doi.org/10.1145/1140123.1140196

Qian, Y., Lehman, J. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.*, 18(1). https://doi.org/10.1145/3077618

Sooriamurthi, R. (2009). Introducing Abstraction and Decomposition to Novice Programmers. *SIGCSE Bull.*, 41(3), 196–200. https://doi.org/10.1145/1595496.1562939

Sorva, J., Sirkiä, T. (2010). UUhistle: A Software Tool for Visual Program Simulation. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. Association for Computing Machinery, New York, NY, USA, pp. 49–54. 9781450305204. https://doi.org/10.1145/1930464.1930471

Statter, D., Armoni, M. (2020). Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Trans. Comput. Educ.*, 20(1). https://doi.org/10.1145/3372143

Steimann, F. (2018). Fatal Abstraction. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Association for Computing Machinery, New York, NY, USA, pp. 125–130. 9781450360319. https://doi.org/10.1145/3276954.3276966

Wing, J.M. (2006). Computational Thinking. *Commun. ACM*, 49(3), 33–35. https://doi.org/10.1145/1118178.1118215

P. Haglund is a PhD student and a lecturer in the Department of Computer and Information Science at Linköping University, Sweden. He has been teaching computer science at the university since 2018, primarily working with courses in introductory imperative programming, object oriented programming and language design for computer science majors. His interest in research is focused around the learning of computer science in introductory courses and how to investigate challenges arising during these courses. Since 2018 he has also worked with the Swedish National Agency for Education, developing several courses and materials in conjunction with an effort to introduce K-12 students to programming. His efforts here have primarily been focused on teaching programming to teachers.

F. Strömbäck is a PhD student and a lecturer in the Department of Computer and Information Science at Linköping University, Sweden. He has been teaching computer science since 2012, primarily teaching courses on concurrency, operating systems, as well as data structures and algorithms. He started his PhD studies in 2018, and his main research interest is teaching and learning concurrency. As a part of this work, he has also studied prerequisites to learning concurrency, for example by co-chairing an ITiCSE working group on prerequisite skills.

L. Mannila is a researcher in computer science education at Linköping University, Sweden. Her research interests include questions related to computational thinking, digital competence and programming at K-9 level, both from a student, teacher and organizational perspective.