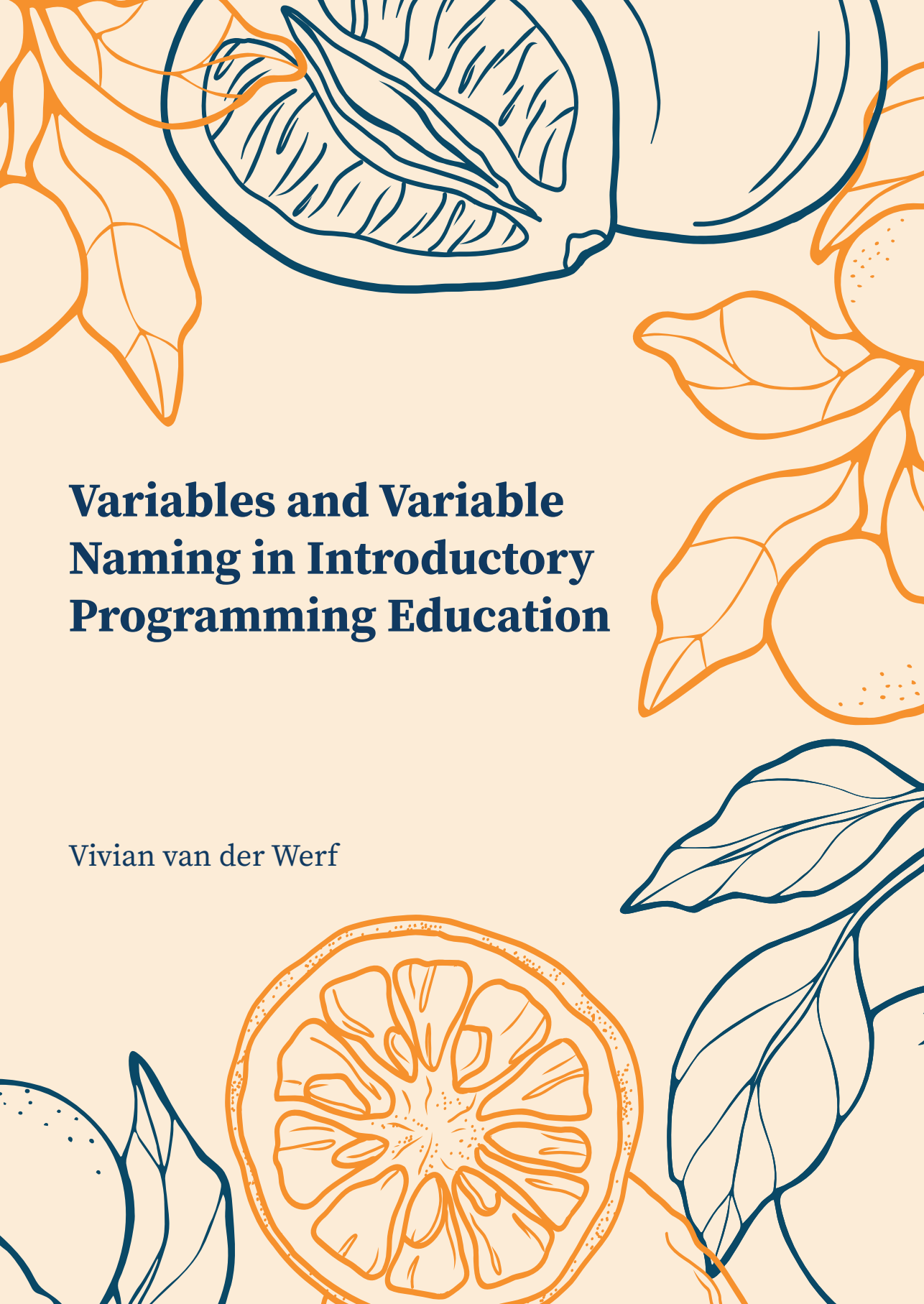
Variables and variable naming in introductory programming education

Werf, V. van der



Variables and Variable Naming in Introductory Programming Education

Vivian van der Werf

Variables and Variable Naming in Introductory Programming Education

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 2 september 2025
klokke 13.00 uur

door

Vivian van der Werf

geboren te Tilburg
in 1992

Promotores:
Prof.dr. M.M. Specht (Leiden University; Delft University of Technology; University of Hagen)
Prof.dr.ir. F.F.J. Hermans (Vrije Universiteit Amsterdam)

Co-promotor:
Dr.ir. E. Aivaloglou (Delft University of Technology)

Promotiecommissie:
Prof.dr. M.M. Bonsangue
Prof.dr. S. Verberne
Dr. M.J. van Duijn
Prof.dr. E. Barendsen (Radboud University Nijmegen; Open University of the Netherlands)
Prof.dr. T. Kohn (Karlsruhe Institute of Technology for Computer Science Education)

ISBN : 978-94-6496-445-5
Cover : Wendy Bour-van Telgen
Printer : Gildeprint

The research in this thesis was performed at the Leiden Institute of Advanced Computer Science (LIACS, Leiden University) and the Leiden-Delft-Erasmus Centre for Education and Learning (LDE-CEL, Delft University of Technology).

Contents

Chapter 1

Introduction

One of my students said to me: *"Miss, if naming already is an issue, we have a big problem!"* This quote comes from a whole-class discussion with young software developers-to-be, where the students debated about whether the topic of naming is an issue for software developers. For me, this statement perfectly reveals two important topics that illustrate the foundation of my research: (1) the relevance of naming practices to software developers and programmers, and (2) whether or not naming practices are hard to learn and apply for students and professionals.

However, before diving in deep, I will first highlight the context of this study. As the interest in programming skills is increasing in society, programming will inevitably become part of national curricula, and in some countries it already is. Through this dissertation, I aim to make programming more accessible to everyone, and contribute to inclusion of students and teachers from diverse backgrounds in programming education. Indeed I hope to inspire programming teachers to expand their focus beyond mathematics and problem solving. In particular, I approach the learning and teaching of a programming language from a natural language perspective. In this approach, I see natural language, that is already familiar to students, serving as a bridge between complex programming problems and the programming language itself.

Moreover, neuroscientific results indicate that language-related brain areas are more important for programming than brain networks related to problem-solving or mathematics [Prat et al., 2020, Endres et al., 2021b, Floyd et al., 2017]. Furthermore, classroom experiments demonstrate how training on technical reading abilities can improve programming skills [Endres et al., 2021a] and how reading code aloud, a strategy derived from natural language learning, helps with remembering syntax, which is an element of coding that beginners often struggle with [Hermans et al., 2018a, Swidan and Hermans, 2019]. Research on reading and explaining code has also shown that code reading skills correlate to code writing skills [Murphy et al., 2012a, Whalley et al., 2006].

From prior work, we can thus establish that learning a programming language is related to natural language and natural language skills. Keeping this in mind, my research focuses on one element of natural language found in written code, in particular, names, also known as identifiers or identifier names. Names are assigned to objects like variables and functions, among others –or the memory addresses associated with the objects– to represent a 'label' for the human reader, helping them remember what the variable refers to. For the non-programmer, these identifiers have a very similar function to labels or tags to represent the contents of, for example, a moving box.

Software engineering research has repeatedly confirmed the importance of identifier naming in reading and understanding code [Lawrie et al., 2006, Feitelson, 2023, Avidan and Feitelson, 2017, Hofmeister et al., 2017, Hofmeister et al., 2017, Schankin et al., 2018, Arnaoudova et al., 2016, Feitelson et al., 2022]. The main takeaway from these works is that programmers rely on names for their comprehension of code. In particular, research indicates that good names support a programmer's comprehension while bad names interfere with their understanding or can even be (unintentionally) misleading. This means that programmers may end up with an incorrect understanding of a program due

to naming issues, which makes them slower in writing code, understanding its function, or finding a bug. Especially general, non-specific names, such as 'length' [Feitelson, 2023] or 'result' [Schankin et al., 2018], appear problematic, as well as context-less letters and unclear abbreviations [Lawrie et al., 2007b, Lawrie et al., 2006, Hofmeister et al., 2017, Beniamini et al., 2017], or names that are too long or too similar to remember well [Binkley et al., 2009]. To make it worse, also full words can be unintentionally misleading, depending on context and the interpretation of the reader [Avidan and Feitelson, 2017, Arnaoudova et al., 2016, Feitelson, 2023, Feitelson et al., 2022]. It is safe to say that any name should be chosen carefully and cautiously.

In practice, programmers generally follow standard conventions with clear rules on, for example, when (not) to capitalize letters (like the *Elements of Java Style* and *Java Language Specification*) [Butler et al., 2015]. They furthermore use a limited vocabulary [Lawrie et al., 2006, Antoniol et al., 2002, Caprile and Tonella, 1999] and commonly use single letter names, like *i, e, s*, and *c*, which cover one-tenth to one-fifth of all names in C, Java and Perl projects [Beniamini et al., 2017, Gresta et al., 2021]. The most popular names are non-specific names such as 'value', 'result', and 'name'. Other names follow a *singular noun-phrase pattern* such as 'nextArea' or 'max_buffer_size', or are formulated as plural when they concern a certain collection (of lists, arrays, etc.) or data grouping [Newman et al., 2020]. In Scratch projects, a block-based programming language originally directed at children, names tend to be longer than in other languages with only one in twenty-five names covering just single letters like *i, x*, and *y* [Swidan et al., 2017].

While different names *can* be understood by the majority of developers [Feitelson et al., 2022], developers still choose to rename, often to narrow the meaning and support code comprehension [Peruma et al., 2018]. Moreover, one in four code reviews contains suggestions about naming [Allamanis et al., 2014], indicating that the original naming was often not clear enough or perhaps incorrect. Professional guidelines describe good naming as 'meaningful', 'clear', and 'concise', mention to use 'familiar names' within the domain, and use words that are 'present in a dictionary' (i.e., [Vermeulen et al., 2000]). This shows that some of the research on (good) naming practices has been incorporated. However, what these statements and suggestions mean in more detail – in other words, how names are to be chosen or how 'meaningful' naming is to be applied in which context – largely remains up to the developer's interpretation of the guidelines, the context, length, and purpose of the code, and the developer's creativity or professional requirements. Hence, choosing appropriate names is much less straightforward than many guidelines make it seem, or many programmers may think, even for professional developers.

As the research shows, developers are still choosing non-specific names that hinder code comprehension and remain affected by naming choices. This begs the question of how novices and learners are affected by naming practices that they encounter, especially if they also have not learned the meaning of certain single-letter names, which might be common and obvious to professionals. Although research has yet to investigate how exactly such naming practices influence learners *or* learning, I will go ahead and assume that all naming practices affect novice programmers more than experienced professionals. I am confident in making this assumption for two reasons. First, novices are easily overwhelmed by the many new aspects that learning a (new) programming language brings [Hermans, 2020], which pressures the cognitive load of students. As a consequence, they might be

even more dependent on natural language for the comprehension of programs or while learning unfamiliar programming constructs. Using good names could thus facilitate learning, while bad names may handicap them or even lead them astray. Second, novices may still hold certain misconceptions, such as wrongly believing that computers interpret or assign values based on the semantic meaning of variables' names, which leads them to incorrectly apply semantic assumptions to syntax [Kaczmarczyk et al., 2010].

Time to come back to the opening quote; *"Miss, if naming already is a problem, we have a big issue!"*. Logical reasoning shows that *if* naming practices are highly relevant to programmers (they are), *and* naming practices cause issues among students and professionals (they do too), *then* the topic is important within programming education and deserves appropriate attention from the community. Unfortunately, and in stark contrast to the extensive research on the effect of names on programming comprehension, very little research covers the topic of naming practices in programming education. Some efforts have been made to incorporate naming practices in rubrics for teaching code quality and assess students' assignments [Stegeman et al., 2014, Stegeman et al., 2016, Glassman et al., 2015]. Indeed, feedback on naming practices with good *and* bad examples is highly valued by students [Glassman et al., 2015], and feedback related to code quality is frequently asked for [Börstler et al., 2017]. This suggests that topics such as code quality and naming practices, might not get enough dedicated attention in educational settings, however, comprehensive investigations into this research area are lacking in the existing literature.

1.1 Research objectives

This dissertation aims to open a scholarly discussion on naming practices in programming education, examining how these practices are, can, or should be effectively used and integrated in teaching novices. It furthermore aims to provide practical advice for educators on how to incorporate naming practices in their courses to enhance understanding, improve code readability, and shape the development of future programming curricula. Hence, my work strives to influence practitioners in the fields of Computer Science and Software Engineering as well as those involved with teaching programming skills to novices and professionals.

Before diving into the topic of naming, my dissertation starts with a wider exploration of code comprehension through reading. In particular, the following research question is addressed first:

RQ1 What do novice programmers express in their answers when asked to explain given code segments in their own words?

Then, to contribute to a scholarly discussion and inform educators on the topic of naming practices in programming education, my research investigates how teachers perceive naming practices, how the topic is currently taught, and how the topic should be taught based on scientific evidence. Therefore, this dissertation also addresses the following research questions:

RQ2 How are variables and their naming practices introduced in beginner programming education and materials?

RQ3 What are teachers' beliefs and perceptions about naming practices and teaching them?

RQ4 How can we incorporate activities that focus on naming in beginner programming education?

1.2 Research design and sneak preview

To answer the research questions, my research implements a qualitative and exploratory approach and uses different types of data. **Table 1.1** shows an overview of the different studies and my research approach is further detailed below.

First, to investigate how novice programmers make sense of code reading exercises **(Chapter 2)**, I present students with short programs (also containing natural language) and ask them to explain what the code does in their mother tongue. Such tasks are also known as *explain-in-plain-english* (EiPE) tasks. To find patterns in these explanations **(RQ1)**, I perform an exploratory **artifact analysis**, addressing three aspects: the explanations' focus, which elements are (not) included, and whether any misconceptions are demonstrated. Among the findings was that students rely on the available natural language that is presented in print and input statements, and names of variables and functions. Particularly relevant to this thesis, I found that students are influenced (or distracted) by such natural language in their interpretation of a program's purpose. These findings highlight the importance of natural language within a code and piqued my interest in variable naming practices.

Then, to explore the current landscape of teaching approaches and learning activities that focus on variable naming practices **(RQ2; Chapter 3, Chapter 4, Chapter 5)**, I start by **interviewing** teachers about their perceptions of variable naming in general **(RQ3)**, their beliefs about good naming practices **(RQ3)**, and their approaches to teaching the topic **(RQ2) (Chapter 3)**. These interviews are analyzed through an **open-coding** process and reveal self-reported approaches. To confirm and extend these self-reported approaches, I also **observe** actual teaching practices and educational materials in popular online courses (*Massive Open Online Courses: MOOCs*) **(Chapter 4)** and programming textbooks for children and novices **(Chapter 5) (RQ2)**. All three studies consistently reveal a wide variation in how naming practices are taught, with a strong(er) focus on 'syntax rules' demanded by the programming language rather than on what meaningful naming is, or why it is relevant. Among teachers, there is a dominant belief that naming is not difficult and is learned 'naturally by example'. However, the examples that students are shown in course materials, and the explanations that are given to them (if any), are often uninformative and inconsistent. Moreover, feedback is rarely provided and students are not encouraged to pay good attention to naming practices as the emphasis lies on whether the code works. Whether the code is readable or adheres to code quality norms is regarded as secondary, as evidenced by teachers (implicitly or explicitly) and educational materials, which sometimes even deliberately state that naming is not important and you can choose any name you like.

These results reveal that the opportunities that students have to develop good naming practices are limited. Knowing that good naming practices are essential within the

Table 1.1: Overview of studies

	RQ	Focus			Data source(s)	Type of (qualitative) research *(output or theme)*
		Students	Teachers	Materials		
Ch. 2	1	x			code explanations	artifact analysis (*code comprehension, EiPE skills*)
Ch. 3	2,3		x		interviews	open-coding (*current perceptions, teaching approaches*)
Ch. 4	2			x	MOOCs	observation (*current educational content, teaching approaches*)
Ch. 5	2			x	textbooks	observation (*current educational content, teaching approaches*)
Ch. 6	4	x	x	x	workshop, interviews	design, implementation, analysis (*learning activities, guidelines*)

programming profession, my studies thus demonstrate an opening for a better implementation of naming practices in programming education. In particular, I see room for interactive activities that focus on discussing why naming is relevant and when is a name meaningful, rather than telling students that naming is important, referring them to guidelines, or focusing on a set of specific rules that might differ per programming language or context.

Hence, to inform educators on how they can tackle the topic of naming practices in their courses **(Chapter 6)**, I present the **design** of a set of learning activities **(RQ4)** following a *dialogic teaching approach*. These activities emphasize reflection and discussion and provide easy-to-implement opportunities for students to see and discuss the effect of different naming choices. For example, I present a ranking activity focused on a set of names which encourages reflection because it requires students to rely on their perceptions and opinions to evaluate what they consider appropriate or misleading. Discussing these rankings (students' opinions) in class provides an opportunity for students to experience that naming needs are not as straightforward as they seems at first sight. Through **implementing** and **testing** the designed activities **(RQ4)**, I determine several insights and recommendations regarding the adoption of naming practices in a curriculum, such as the importance of whole-class discussion and individual reflection, and the easy adaptation of activities to fit any course without much teacher investment. Moreover, the activities reveal potential issues and obstacles perceived by the students, such as that paying attention to naming is considered too time-consuming, inefficient, or even irrelevant, even when the importance of naming for a (second) reader is recognized by the students. These findings highlight the relevance of 'priming' students to adopt good naming practices before expecting them to 'figure it out themselves'.

1.3 Chapter outline

In the above section, I presented some of the chapters' highlights and how each study shaped my research journey. In this section, I outline the structure of the dissertation by providing an overview of each chapter.

Chapter 2 introduces the importance of code reading exercises in learning a programming language. In particular, EiPE tasks are discussed, as well as code comprehension in general. The research presented in this chapter provides insight into what novice students express in their explanations after reading a piece of code, and what these insights reveal about how the students comprehend code. I performed an **exploratory** analysis on four reading assignments extracted from a university-level beginner's course in Python programming and paid specific attention to (1) the core focus of student answers, (2) elements of the code that are often included or omitted, and (3) errors and misconceptions students may present. I found that students prioritize the output that is generated by print statements in a program, followed by control flow elements and function definitions. Some students omit (relevant) details on the code's purpose beyond the information conveyed through natural language, and their explanations are negatively affected when these names convey unhelpful or distracting information. This shows that students rely on natural language elements of the code when they are asked to explain a program, which shows that explaining a program does not necessarily mean that they have understood the code.

Chapter 3 introduces the importance of variable naming practices for code writing, comprehension, and debugging, while at the same time demonstrating that little is known about how variable naming is taught. The research presented in this chapter investigates naming beliefs, self-reported teaching practices, and observations regarding variable naming practices of teachers of introductory Python programming courses. I adopted an in-depth qualitative approach by **interviewing** ten teachers from secondary education and higher education and developed several themes to answer our research questions. Among various opinions and practices, I found that teachers agree on using meaningful names, but have conflicting beliefs about what is meaningful. Moreover, the described teaching practices do not always match teacher's views on meaningful names, and teachers rarely encourage students to use them. Instead, teachers express that naming practices should not be enforced and that students will develop them by example. Whereas some teachers report focusing solely on conventions, others deliberately dedicate time for students to engage with naming, create self-made guidelines, provide continuous feedback, or include naming exercises on exams. This chapter concludes that naming practices are not deliberately taught even though they influence program understanding and code quality, as there exist inconsistencies in teachers' self-reported naming practices.

Chapter 4 and **Chapter 5** focus on the concept of variables in general and on variable naming practices, and aim to understand how these are introduced in *Massive Open Online Courses* (MOOCs) **(Chapter 4)** and programming textbooks for children and adults **(Chapter 5)**. The research presented in these chapters investigates (1) which definitions and analogies are currently being used to explain the concept of variables, (2) which programming concepts are introduced alongside variables, and (3) if and how variable naming practices are introduced in the materials. To answer these questions, I gathered qualitative data related to variables and their naming by **observing** 17 MOOCs (Java, C, Python) and by analyzing 13 programming textbooks (Python, Scratch). Collected data include connections to other programming concepts, formal definitions and used analogies, and explanations and examples used to introduce variable naming practices. I found that analogies are often explained using the 'variables-as-a-box' analogy, although some books also introduce them as a 'place' or 'label'. The definition of a variable mostly focuses on storing information whereas other elements such as tracking or accessing information, computer memory, or flexible use and changing values remain underrepresented. I furthermore found differences between programming languages in the order in which variables and other concepts are introduced, but the most connected programming constructs are data types, program execution/control flow constructs, and operators/expressions. Finally, in both MOOCs and textbooks, I found inconsistent teaching of naming practices that focus on *syntax rules* which, when not adhered to, break the program. Most courses and books remain vague about –or display disagreeing notions on– 'what is a meaningful name', and present only a few examples of good and bad names. These observations match the teachers' self-reported approaches and perceptions, meaning that there is room for more deliberate attention to the meaning of variable names within the current landscape of teaching naming practices.

Chapter 6 addresses how to teach naming deliberately, without centralizing specific naming rules or styles, and instead focusing on discussing questions such as why is naming important and when is naming meaningful. The chapter presents a **dialogic teaching**

approach focused on teaching a critical reflection on naming practices through the **design** of five types of activities: (A) expressing perceptions and experiences, (B) creating names, (C) evaluating names through ranking, (D) comparing codes, and (E) locating a mistake. For this study, I developed, ran, and analyzed a one-hour workshop, which is presented here together with the experiences gained by teaching it to two courses. Ultimately, this chapter leads to recommendations for teachers and has a two-fold contribution: (1) a set of (adaptable) activities and exercises for supporting deliberate naming practices that assist teachers interested in adopting them into their curriculum; (2) insights regarding the student perspective on naming practices, derived from the activities, revealing potential issues and opportunities in teaching the topic.

Chapter 7 presents the general discussion of this thesis by highlighting several key findings, placing them in a wider context, and discussing relevant implications for both educators and academics within the field of Computer Science Education. The chapter finishes with a comprehensive summary of the dissertation's main conclusions and a list of my further recommendations for future research and educational practice.

1.4 Origin of chapters

All chapters of this thesis have been published as full papers in peer-reviewed conferences. Chapters Two, Three, Four, and Five are all empirical studies, and Chapter Six is published as an experience report. Besides formatting, no changes were made to the papers' original text or content.

Parts of this **introduction** are based on (1) a poster abstract and presentation at the conference of International Computing Education Research (**ICER'22**) in Lugano, Switzerland, titled *(How) Should Variables and Their Naming Be Taught in Novice Programming Education?*, by Van der Werf, Aivaloglou, Hermans, and Specht [van der Werf et al., 2022a]; and (2) a doctoral consortium poster and presentation at **KoliCalling'23** in Koli, Finland, titled *Fostering a natural language approach in programming education (Doctoral Consortium)*, by Van der Werf [van der Werf, 2024].

Chapter 2 was published as *What does this Python code do? An exploratory analysis of novice students' code explanations*, by Van der Werf, Aivaloglou, Hermans, and Specht, in the Proceedings of the 10th Computer Science Education Research Conference (**CSERC'21**), and presented online in 2021 [van der Werf et al., 2022b].

Chapter 3 was published as *Teachers' Beliefs and Practices on the Naming of Variables in Introductory Python Programming Courses*, by Van der Werf, Swidan, Hermans, Specht, and Aivaloglou, in the Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (**ICSE-SEET'24**), and presented in Lisbon, Portugal in 2024 [van der Werf et al., 2024c].

Chapter 4 was published as *Variables in Practice. An Observation of Teaching Variables in Introductory Programming MOOCs*, by Van der Werf, Zhang, Aivaloglou, Hermans, and Specht, in the Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education (**ITICSE'23**), and presented in Turku, Finland in 2023 [van der Werf et al., 2023].

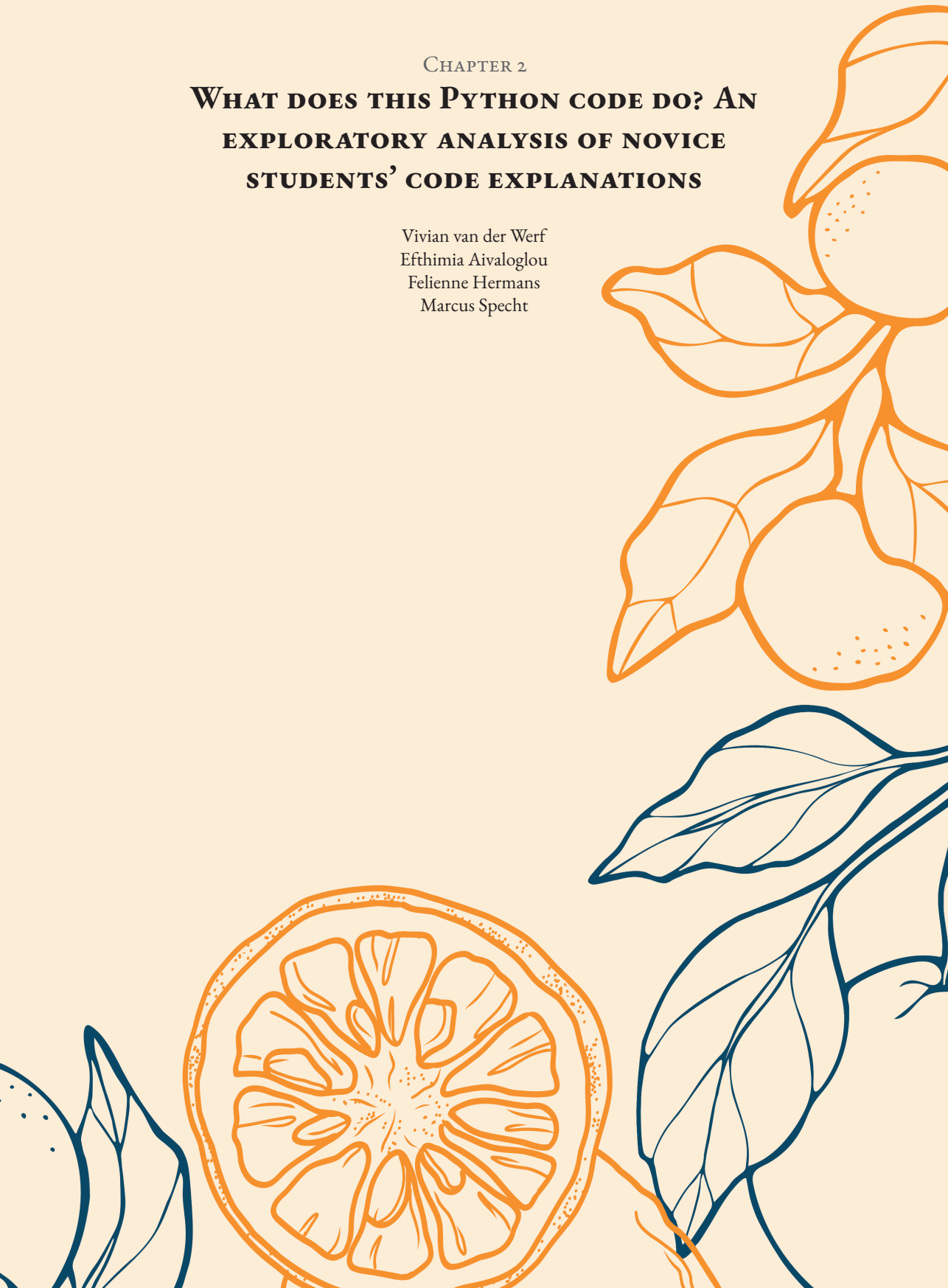
Chapter 5 was published as *Variables and Variable Naming in Popular Programming Textbooks for Children and Novices*, by Van der Werf, Hermans, Specht, and Aivaloglou, in the Proceedings of the 2024 ACM Virtual Global Computing Education Conference (**SIGCSE Virtual'24**), and presented at the online venue in 2024 [van der Werf et al., 2024b].

Chapter 6 was published as *Promoting Deliberate Naming Practices in Programming Education: A Set of Interactive Educational Activities*, by Van der Werf, Hermans, Specht, and Aivaloglou, in the Proceedings of the 2024 ACM Virtual Global Computing Education Conference (**SIGCSE Virtual'24**), and presented at the online venue in 2024 [van der Werf et al., 2024a].

Chapter 2

What does this Python code do? An exploratory analysis of novice students' code explanations

Vivian van der Werf
Efthimia Aivaloglou
Felienne Hermans
Marcus Specht



Abstract

Code reading skills are important for comprehension. Explain-in-plain-English tasks (EiPE) are one type of reading exercises that show promising results on the ability of such exercises to differentiate between particular levels of code comprehension. Code reading/explaining skills also correlate with code writing skills. This chapter aims to provide insight into what novice students express in their explanations after reading a piece of code, and what these insights can tell us about how the students comprehend code. We performed an exploratory analysis on four reading assignments extracted from a university-level beginner's course in Python programming. We paid specific attention to (1) the core focus of student answers, (2) elements of the code that are often included or omitted, and (3) errors and misconceptions students may present. We found that students prioritize the output that is generated by print-statements in a program. This is an indication that these statements may have the ability to aid students make sense of code. Furthermore, students appear to be selective about which elements they find important in their explanations. Assigning variables and asking input were less often included, whereas control-flow elements, print statements, and function definitions were more often included. Finally, students were easily confused or distracted by lines of code that seemed to interfere with the newly learned programming constructs. Also, domain knowledge (outside of programming) both positively and negatively interfered with reading and interpreting the code. Our results pave the way towards a better understanding of how students understand code by reading and of how an exercise containing self-explanations after reading, as a teaching instrument, may be useful to both teachers and students in programming education. [1]

Keywords

program comprehension
CS education
Python
code reading
EiPE
qualitative content analysis

[1] Published as: van der Werf, V., E. Aivaloglou, F. Hermans, and M. Specht (2021). *What does this Python code do? An exploratory analysis of novice students' code explanations*. In Proceedings of the 10th Computer Science Education Research Conference, **CSERC 2021**, page 94–107, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3507923.3507956

2.1 Introduction

Historically, programming education has predominantly focused on writing code to teach programming concepts and program understanding [Busjahn and Schulte, 2013, Izu et al., 2019]. As code writing is seen as the most complex programming skill, it is often assumed that when you can write a program, you understand how it works [Salac and Franklin, 2020]. However, recent research suggests that lower level programming skills, such as tracing and reading, are at least equally important for novice programmers, since students' mastery of these skills correlates with their code writing ability [Corney et al., 2011, Lister et al., 2009, Lopez et al., 2008, Venables et al., 2009]. Moreover, Lethinen et al. [Lehtinen et al., 2021a] found that even students who correctly write programs struggle with explaining their own code. Such findings underpin the potential of reading exercises for learners. Furthermore, reading exercises may also encourage teachers who are not (or are less) familiar with code writing themselves. Reading exercises may be more recognizable to them, as these exercises are able to mimic teaching strategies from other disciplines, such as math and language. Therefore, teachers may also require less deep initial understanding of programming. However, reading exercises are not widely implemented in programming education yet [Fowler et al., 2021, Busjahn and Schulte, 2013, Izu et al., 2019].

This chapter explores the act of reading and explaining code with the help of "Explain in plain English" (EiPE) exercises. EiPE exercises are one particular way to practice and evaluate code reading skills and grew in popularity in research on programming education during the last decade. Research has confirmed the immense potential of these exercises in developing and strengthening novices' programming skills. Most research focused on one of two aspects: (1) the SOLO ("Structure of the Observed Learning Outcome") taxonomy to evaluate and assess student answers, often in relation to other programming skills [Biggs and Collins, 1982, Clear et al., 2008, Corney et al., 2014, Lister et al., 2006, Sheard et al., 2014, Whalley and Kasto, 2014, Whalley et al., 2006] or (2) other frameworks to rate the answers in regards to comprehension [Chen et al., 2020, Weeda et al., 2020]. Both aspects center on evaluating comprehension from reading. However, in this chapter we aim to explore how students think about code when they are learning new programming concepts. Hence, we gather information outside such assessment frameworks, something that, to our knowledge, has not yet been documented within previous research on EiPE-questions. This means that, rather than using a fixed model or framework as a spyglass to look at student's answers, we analyze their answers in an open-ended, exploratory way. To this end we mainly focus on what students take away from reading a piece of code, and are less interested in how well students comprehend that code after reading. After all, this has already been intensively covered by prior works. Our assumption is that information on what students express when explaining code can reveal information about students' comprehension processes.

In particular, we analyze in an exploratory manner what happens when we ask novice students to explain a piece of code in their own words. The research question central to this chapter is *what do novice programmers include in their answers when asked to explain given code segments in their own words (plain English)?* To answer this question, the following sub questions are relevant:

RQ1 What is the core focus of the explanations?

RQ2 What elements are most present and which are absent in the explanations? (e.g. lines of code or particular programming concepts)

RQ3 What types of mistakes or misconceptions are demonstrated by the explanations?

2.2 Background and related work

2.2.1 Code comprehension

When writing code, programmers construct their own mental models about the code and its programming concepts. This process is usually referred to as program comprehension [Izu et al., 2019] and much research has been done around this topic. Recently, program comprehension is increasingly recognized as important during learning, to improve students' overall coding skills [Izu et al., 2019]. Tasks that foster program comprehension usually pertain reading, interpreting and explaining code, as well as tracing, editing, debugging or extending existing code. Moreover, tasks like tracing and reading code could provide novices with better opportunities to practice difficult concepts, as these activities usually take less cognitive load than code writing [Busjahn and Schulte, 2013]. Too much cognitive load prevents students from learning. Xie et al. [Xie et al., 2019] therefore argue that students should practice understanding common code patterns by reading first, before attempting composing these patterns in writing tasks.

There exist many theories, models and frameworks concerning program comprehension in education, which are well discussed in [Izu et al., 2019]. However, one well-establish framework for evaluating code comprehension in education is Schulte's Block Model [Schulte, 2008], which can be used to analyze how novices make inferences when trying to comprehend a code. The model differentiates between the types of information in a code (text surface, program execution (e.g. data flow), program goals) and the size of the entities in a code (atoms, blocks, relations, macro structure) (see **Table 2.1**). Schulte suggests that understanding a program means to be able to build a bridge between the lowest forms of either dimension (text:atom) and the highest forms of either dimension (goals:macro). This includes a translation from the technical structure of a program to its social function. Such translation often causes a learning problem because students can have a limited understanding of the structure. Students can also have limited understanding resulting from the code's structure itself, since from the structure there exists no direct path leading to function. Moreover, social functions can often be interpreted differently, leading to miscommunication about the program [Schulte, 2008]. The block model thus highlights the need for translation between code and function for comprehension. During this process, all its different levels play a specific role in program comprehension. It is, therefore, no surprise that the Block Model framework is regularly used as a foundation to investigate or assess code comprehension. In this chapter, the Block Model serves as an example of a means to assess code comprehension in general and is therefore an interesting perspective to some specific results of this work.

Table 2.1: Schulte's Block Model, after [Lehtinen et al., 2021b, Schulte, 2008]

	Text – *technical structure*
Atom	language elements
Block	syntactically/semantically related elements
Relational	connections between "blocks"
Macro	entire program
	Execution – *technical structure*
Atom	elements' behavior
Block	a "block's" behavior
Relational	flow between "blocks"
Macro	the program's behavior
	Goals – *social function*
Atom	elements' purpose
Block	a "block's" purpose, program subgoal
Relational	integration of subgoals
Macro	the program's purpose

2.2.2 Assessing comprehension

It is generally assumed that there exists a certain hierarchy in learning to code [Lister, 2016, Lister, 2020, Xie et al., 2019, Busjahn and Schulte, 2013], one that is not unlike learning a (foreign) language: knowing the syntax, being able to trace code, being able to read code and abstract beyond the code, and finally, being able to write code. Just like writing a well-reasoned essay usually confirms language abilities, writing a program is often seen as the capstone of programming skills: if you can write a program yourself, you are considered a programmer and it is assumed you have demonstrated the skills that are lower in the mentioned hierarchy.

However, recent research by Salac and Franklin [Salac and Franklin, 2020] on the relationship between 'artifact analysis' (analyzing programs created by students) and summative written assessments in introductory computing, using Scratch as case-study, has observed only a weak link between them. This suggests that artifact analysis does not measure whether a student truly understands their written code, leaving Salac and Franklin to conclude that code-writing assignments are *"an expedient but inaccurate choice"* for measuring code comprehension [Salac and Franklin, 2020]. A think-aloud study by Kennedy and Kraemer [Kennedy and Kraemer, 2019] also found that students write "working" code through a trial-and-error strategy, without them actually understanding (or using) concepts that were to be learned.

In other words, students that write code may not understand all its programming constructs. Vice versa, students that understand certain programming concepts may choose not to use them in their own programs. This conclusion supports earlier work by Brennan and Resnick [Brennan and Resnick, 2012], who concluded that assessment should not only focus on product-based assignments, but also incorporate computational thinking processes and computational thinking perspectives into the evaluation of the

(developing) computational thinker.

Finding a good indicator of a student's skills on program comprehension proves to be difficult. Perhaps reading comprehension assignments, such as EiPE tasks, can be an efficient, complementary alternative to traditional code writing assignments (see also [Salac and Franklin, 2020]). Various researchers [Chen et al., 2020, Corney et al., 2014, Murphy et al., 2012a, Whalley et al., 2006] reported relative strong correlations between reading and writing exercises, as well as between reading exercises and overall performance. Furthermore, correct explanations about one's own code also seems to correlate with increased success [Lehtinen et al., 2021a]. These findings indicate that EiPE exercises are, in fact, promising when it comes to evaluating program comprehension. However, it is still unclear what causes this relation. Are students better at writing code because they can better abstract from code while reading, and thus better explain it? Are they better at abstraction because they know how to write a program? Or are there perhaps different skills at play that increase both reading (abstraction/explanation) as well as writing skills? Such questions still remain open.

2.2.3 EiPE exercises

Weeda and colleagues [Weeda et al., 2020] describe the idea of an Explain-in-Plain-English (EiPE) task as to summarize the goal of a given code. It is assumed that *"students who comprehend a program (or code segment) should be able to provide a clear and coherent description of its overall purpose as a whole, beyond merely tracing its execution or providing a line-by-line description"* [Weeda et al., 2020]. This definition shows the general direction of an EiPE task across literature, where the task serves to assess a student's functional understanding of a piece of code (see also [Fowler et al., 2021, Murphy et al., 2012a, Salac et al., 2020, Corney et al., 2014, Murphy et al., 2012b, Corney et al., 2012, Pelchen and Lister, 2019, Chen et al., 2020]). Note that, in order to measure this functional understanding (i.e. comprehension), the intended goal of this task is usually to summarize the purpose of a code, without including a line-by-line description.

Research indicates that EiPE questions have been proven to effectively differentiate between students who summarize code with a high level of abstraction beyond the code (also known as a "relational answer") and those that do not [Chen et al., 2020, Corney et al., 2014, Murphy et al., 2012a, Pelchen and Lister, 2019, Weeda et al., 2020, Whalley et al., 2006, Corney et al., 2012]. Moreover, students who provide the general purpose of the code in such EiPE exercises, score better on other types of programming exercises as well, such as code production exercises [Corney et al., 2014, Murphy et al., 2012a, Sheard et al., 2014, Whalley et al., 2006, Chen et al., 2020]. Corney et al. [Corney et al., 2011] also found that when students have difficulties explaining their code in terms of it's purpose early in the semester, they struggle with writing code later that semester.

Additionally, Pelchen and Lister [Pelchen and Lister, 2019] compared relational answers on twelve different EiPE-exercises and concluded that they can be used as an indicator for code comprehension. They studied the frequency of words used in the answers given by novice programmers, and found statistically significant differences in word use and word frequency between those students who answered all questions correctly, and those who did not. Specifically, the first group seemed to be more precise, more comprehensive and

more likely to mention words that were an abstraction beyond the explicit code [Pelchen and Lister, 2019]. These results underpin the ability of EiPE-exercises to differentiate between the comprehension level of different students. This in turn paves way to its use in the assessment of program comprehension, and, as a result, they can be a potential goldmine for understanding how students read and explain code. Consequently, reading exercises, such as EiPE tasks, should be able to provide information about how students develop their programming skills.

2.3 Methods

In order to gain further insights in how novice students learn and comprehend code, the aim of the current chapter is to investigate what information students present when they explain code segments in plain English. For this purpose, this research investigates multiple EiPE reading exercises that were extracted from an introductory Python programming course. The course, its participants and the investigated materials will be discussed below.

2.3.1 The course: setting and participants

The exercises that we analyzed in this chapter are part of a 12-week CS1 (bachelor level) course at Leiden University, The Netherlands. This course has been running for four years. The course and all its exercises were provided via Stepik, an online learning platform that allows teachers to combine video lectures with different kinds of (coding) exercises that can be graded automatically. The course included information recall questions, recognizing and trying code, writing exercises, reading exercises and reflection exercises.

All assignments, as well as the video lectures that introduce new topics and explain or analyze difficult concepts step-by-step, were available to the students from home for self-learning. Additionally, once a week the students could come to physical class (1.5 hours) to work on the course by themselves, with classmates, or with help of trained teaching assistants. This way it was possible for students to ask questions or get guidance when they found the materials challenging. Every week new topics were introduced, and previous topics could be practiced. The course philosophy has been based on direct instruction [Kirschner et al., 2006, van Merriënboer and Kirschner, 2013], considering cognitive load [Kirschner et al., 2006, Lister, 2016, van Merriënboer and Kirschner, 2013, Sweller, 2011], retrieval practice and reflection to give shape to the course.

The course ran during the last quarter of 2020 (Sept-Dec), with examination in January 2021. It was part of the mandatory curriculum for the BSc Computer Science at Leiden University and included students specializing in informatics, bioinformatics and economy & informatics. Simultaneously, the course was also provided as an Honours program elective for excellent students from different (science) backgrounds and was open for other interested students and individuals. For this chapter, we included data from all individuals participating in the course (N=182). At the beginning of the course, questions were asked on prior knowledge of, and experience with, programming so that the teacher was acquainted with the students' background. It was determined that many students already had some experience with one or more programming languages: 41% indicated they previous experience with Python, 13% with Java, 9% with Scratch and over 27% already used another language prior to this course. Programming concepts that students were

Table 2.2: Participants' self-assessed knowledge of programming concepts (N=178).

	I can recognize	I can write*
Variables	78%	58%
Loops	58%	33%
Functions	67%	37%
Boolean equations	30%	17%
List comprehension	13%	9%
Classes	29%	12%
Functional programming	12%	6%

** without consulting Google or other sources.*

already familiar with are shown in **Table 2.2**. Other student characteristics, such as gender and age, were not asked during the course. There is no indication that the population of this course differs greatly from general university-level programming courses, however, since more specific data was not available, no validation could be made.

2.3.2 Investigated materials

As outlined above, each week new topics or programming concepts were introduced (see **Figure 2.1**). Students were encouraged to practice these new topics as well as topics of previous weeks through various (guided and non-guided) assignments that were given. At the end of each week, students applied their knowledge and skills through questions focusing on theory, reading code, producing code (solving problems) and reflection questions. Exercises focusing on reading code included EiPE-questions. All the EiPE questions in the course were designed for the students to get familiar with code reading and practice those skills, as well as to repeat the code concepts they had learned so far. Most of the EiPE questions were followed up by multiple choice comprehension questions and/or open-ended reflection questions to help them (re)read and understand the given code, however, no model-answers (EiPE-explanations) were provided, neither before nor after the exercises. Four of the EiPE questions are selected for further investigation: 1) a simple if-elif-else construction; 2) a simple for loop; 3) a while-loop with nested if-else condition, and 4) a larger (disguised) rock-paper-scissors (RPS) game that consists of a main function and combines multiple functions (see **Figure 2.2a-d**). Their corresponding comprehension and reflection questions are not investigated in this research. Note that although no questions of weeks 6, 7, and 8 were selected for this research, students did get to practice more with similar exercises corresponding to that week's topics, therefore slightly increasing in complexity each week.

All aforementioned assignments were selected because they present different programming concepts in a comprehensive way. The if-else assignment (**2.2a**) was chosen for analysis because it was the first EiPE task the students had seen in the course. It also contains a reference to the Dutch grading system, specifically designed to see whether students are reading the code, or merely depending on their prior knowledge of aforementioned system to interpret the code. This assignment is therefore particularly interesting to answer our question concerning mistakes (RQ3). The for-loop (**2.2b**) was picked because it presents a very short piece of code that solely practices what was learned that week. In

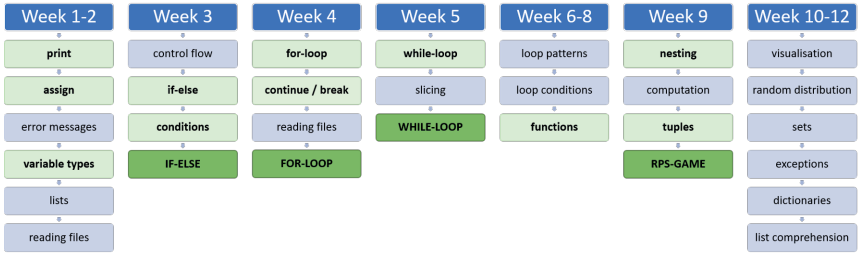


Figure 2.1: Course structure with the programming concepts covered in the course. Light green: first introduction of the programming concepts; dark green: indication of when the reading exercises analyzed in this chapter were introduced.

contrast, the while-loop (**2.2c**) was chosen as it shows slightly more complexity, while remaining a short piece of code, and therefore not too demanding in terms of cognitive level. The disguised RPS-game (**2.2d**) was chosen to include a longer, more difficult exercise towards the end of the course to serve as contrast to assignments 2a, 2b and 2c. The longer code forces students to focus on what they perceive as the most important parts of the code, as a complete line-by-line description would presumably take too much effort. Moreover, this game was intended to be extra challenging for the students and required them to recall and combine knowledge that they had learned so far. However, no new elements were introduced.

Since the exercises were designed to fit the course, they do not directly match EiPE-tasks covered in earlier work (e.g. [Corney et al., 2011, Pelchen and Lister, 2019]). Nevertheless, we believe that the first three exercises contain representative code that is usable for reading exercises such as EiPE, because the type of code segments used are frequently seen in beginner Python courses (e.g. to explain or apply control flow and loops). Similar reasoning applies to the RPS-game: most introductory Python courses include writing or debugging some sort of game and rock-paper-scissers is relatively easy and familiar.

In each of the selected exercises the students were asked to read the piece of code carefully and explain it in "plain English" via a written assignment. It was stressed not to use any jargon (i.e. "if x equals 5 print x else increase x plus 1"). Moreover, it was not important whether the students gave correct or incorrect explanations of the program, nor were students taught to summarize a corresponding purpose. The assignments' primary goal was to help students to read code and reflect on what they can or cannot understand from it. In line with our goal, any other instructions, such as to provide (only) the purpose of the code or not to include a line-by-line description, were not added. Although many works on EiPE-exercises do mention specifically adding such instructions or examples (i.e [Murphy et al., 2012a, Pelchen and Lister, 2019]), we regard this difference as beneficial to our specific research goal: our aim is to explore what happens when you ask a novice programmer to read a code. Asking to summarize the purpose of a code may require additional skills, and such translation from technical structure to social purpose is, according to the Block Model, difficult for novices [Schulte, 2008]. Under those circumstances it is interesting to

```
grade = int(input())

if grade > 10:
  print('You are cheating')
elif grade > 6:
  print('Well done')
else:
  print('Try again')
```

a) if-else (week 3)

```
print("Hello!")

for i in range(5):
  print(i)
```

b) for-loop (week 4)

```
i = 0
while i < 10:
  i += 1
  if i % 2 == 0:
    continue
  print(i)
```

c) while-loop (week 5)

```
import random

human_score = 0
computer_score = 0
human_move = ''
computer_move = ''

def main(human_s, computer_s, human_m, computer_m):
    human_m, computer_m = choose()
    human_m = check(human_m)
    max_win = max_points()
    while True:
        human_s, computer_s = calculate(human_m, computer_m, human_s, computer_s)
        if human_s >= max_win:
            print(str(human_s) + " - " + str(computer_s) + ", the human wins!")
            break
        elif computer_s >= max_win:
            print(str(computer_s) + " - " + str(human_s) + ", the computer wins!")
            break
        human_m, computer_m = choose()
        human_m = check(human_m)

def max_points():
    max_score = int(input("Enter the desired maximum score: "))
    return max_score

def choose():
    choice_pool = ("green", "orange", "purple")
    human_choice = input("Welcome human! Please, enter 'green', 'orange' or 'purple': ").lower()
    computer_choice = choice_pool[random.randint(0, 2)]
    return human_choice, computer_choice

def check(human_m):
    choice_pool = ("green", "orange", "purple")
    check_flag = False
    while not check_flag:
        if human_m in choice_pool:
            check_flag = True
        else:
            human_m = input("Previous input incorrect, please enter your choice again: ")
    return human_m

def calculate(player_m, computer_m, player_s, computer_s):
    if player_m == "green":
        if computer_m == "orange":
            computer_s += 1
            print("I choose orange, I win!!!")
        elif computer_m == "green":
            print("I choose green, it's a tie!!!")
        else:
            player_s += 1
            print("I choose purple, the human wins!!!")
    elif player_m == "orange":
        if computer_m == "orange":
            print("I choose orange, it's a tie!!!")
        elif computer_m == "green":
            player_s += 1
            print("I choose green, the human wins!!!")
        else:
            computer_s += 1
            print("I choose purple, I win!!!")
    else:
        if computer_m == "orange":
            player_s += 1
            print("I choose orange, the human wins!!!")
        elif computer_m == "green":
            computer_s += 1
            print("I choose green, I win!!!")
        else:
            print("I choose purple, it's a tie!!!")
    return player_s, computer_s


main(human_score, computer_score, human_move, computer_move)
```

d) RPS-game (week 9)

Figure 2.2: The four investigated code-snippets. The following assignment was presented to the students: *"Read this program carefully, and then explain in plain English what it does. Try to avoid using jargon as much as possible"*.

analyze how students interpret a "bare" explanation question and what it is they include. After all, students can still choose to add a summary or purpose and it would be interesting to see if this also happens "naturally". Therefore, although the purpose of the tasks used in this study and that of EiPE tasks generally covered in literature may be inherently different, we consider the chosen tasks to be relevant for gathering information on reading and explaining tasks in general as well as specific EiPE tasks.

2.3.3 Data collection, coding and analysis

All student answers were collected in the online environment Stepik. For the purpose of this research, student answers on the selected EiPE questions were downloaded for analysis. Personal data and student characteristics were excluded from the download, making the dataset completely anonymous. Only a student number (given to the students by Stepik) was kept to clean the data when a student provided multiple answers on the same question, and to identify and compare answers from the same student across the four exercises.

Throughout the course the number of unique views per exercise dropped gradually. After a check within the Stepik-environment, this appeared to be the general pattern for all exercises in the course. In theory, all students from the BSc Informatics should have completed the exercises at the time that the data was downloaded. Not completing the exercises would have negatively impacted their grade. The drop in unique views (and thus number of answers) can therefore only partially be explained by students not finishing the course. As the course was open to everyone interested, it is likely that interested students and other individuals (who therefore did not work towards a grade) were dropping out gradually, or working on the assignments on a slower pace. Blank answers (including "I don't know") and nonsense answers (including "this is code") were eliminated from the final dataset as they do not address any explanation of the program. When a student submitted double answers, only the last submitted answer was selected for analysis. The final number of explanations per question can be found in **Table 2.3**. In the end, 58 students answered all four exercises, another 52 students answered three exercises and finally another 72 students answered only one or two exercises.

After removal of blank, double, and nonsense answers, the student explanations were analyzed through inductive and deductive coding by the first author of this chapter. First, the data from each exercise was explored through open inductive coding, after which the emerging categories from each exercise were unified, summarized, and classified, with the research questions in mind. All student answers from the first three exercises (if-else, for-loop, while-loop) where re-analyzed with the new categories and coded deductively.

Table 2.3: Number of explanations analyzed per exercise.

Exercise	Number of explanations analyzed
If-else	175
For-loop	168
While-loop	110
RPS-game	66

For each explanation it was then scored whether a category was fulfilled or not, i.e. was the category present in the explanation? These general categories are: one-sentence-summary, output in words, exact output, and misconception/error. Besides these categories, each exercise also included more detailed categories, coded inductively, that are specific to the presented code; such as the presence of certain code concepts or individual lines of code. All coding was performed by the first author of this work.

The same approach was made for the disguised RPS-game. However, since the nature and complexity of this code is very different from the other three programs, we decided to focus further analysis only on the presence of the one-sentence summary, the output and several code-specific elements. That means that for the RPS-game, mistakes were not included for analysis.

Since the deductive coding was done with the research questions in mind, our questions are answered as follows: RQ1 (focus) is answered with the categories one-sentence-summary, output in words and exact output; RQ2 (inclusion/absence of elements) is answered by using the inductive analysis specific to the different pieces of code; and RQ3 (mistakes) is answered with the help of the category misconceptions/error.

2.4 Results

The research question central to this chapter is *"what do novice programmers include in their answers when asked to explain given code segments in their own words (plain English)?"*. To answer this question, the following sub questions were asked: 1) What is the focus of the explanations, 2) What elements are most present and which are absent, and 3) What types of mistakes or misconceptions are demonstrated by the explanations? The data concerning these sub questions are discussed below. A synthesis per sub question is given, followed by more detailed findings from each of the four exercises.

2.4.1 Focus of the explanations

Our first research question concerns the core focus of the explanations. Our general observation is that, in three of the four exercises, the output generated by print-statements in the program is at the center of the students' explanations. Over 80% of these explanations include the output in words or copy the exact output that would be generated. However, the results from the RPS-game indicate that if the nature of the code is more complex and/or the length of the code is longer, students no longer favor mentioning the generated output and instead shift their attention to various other elements presented in the code. It is possible that the output generated by the first three case-studies was considered "sufficient" to explain the code, whereas for the RPS-game students selected several function definitions as their primary source for explanation. With the RPS-program, students were also more likely to provide an overall summary of the code. More details of these results can be found in the sections below and in **Table 2.4**.

If-else (Figure 2.2a)

The exact text in the print statement for each condition was mentioned by more than 4 out of 5 students (84%). A typical explanation looks like this: *"This program asks the user to type a grade. Then the program checks whether the grade is above 10, if so it prints You are*

Table 2.4: The number and percentage of explanations (per exercise) that contain a one-sentence summary or (some) output generated by the program (in words or exact copy). The categories are not mutually exclusive.

	If-else (N=175)	For-loop (N=168)	While-loop (N=110)	RPS-game (N=66)
One-sentence-summary	29 (17%)	-	34 (40%)	39 (60%)
Output in words	19 (11%)	109 (65%)	87 (79%)	22 (33%)
Copying exact output	147 (84%)	59 (35%)	7 (6%)	5 (8%)
…of which contain no further description	-	10 (6%)	2 (2%)	-

cheating. If the grade is above 6 it prints Well done. If not it prints Try again". Students that did not provide an exact output, all included that an output was given based on the conditions that applied to the grade, for example *"This program will give a reaction for the grade that is inserted"*.

For-loop (Figure 2.2b)

In all answers (100%), the output generated by the print-statements was central to the explanation. One third of the students (35%) also incorporated what they conclude as the exact output in their answers. Examples of these are: *"This program will print 'Hello!' first. After that the for loop will run, printing the numbers 0, 1, 2, 3, 4 each separated by a new line"*, and *"[the program] prints hello and then under that [it] prints 0 1 2 3 4"*. One in six students that mention such exact output explicitly, provide no further descriptions to explain the code (N=10, 6% of total). When the exact output is not included, the output was mentioned slightly more implicitly, like: *"In this program it will first print 'hello!'. Then in the next part it will print the variable i 5 times. The variable i is different in each of the 5 times. It starts at 0, and then every time it will go up one"*, or *"This program prints "hello" and then it prints the numbers 0 to 4 all on new lines"*. Occasionally the explanations only mention *"it will print hello"*, without any referral to the for-loop.

While-loop (Figure 2.2c)

Almost 80% of explanations include the output that is generated by the program's print-statement in words and 6% provides an exact output. The way the output was described, however, differed from student to student. The most common patterns are described below. About one in five explanations (21%, N=23) mention that *"odd numbers"* (or equivalent) are printed by the program. The term "odd numbers" (or equivalent) is mentioned explicitly in three different ways: 1) as part of a stand-alone one-sentence summary, such as *"for the numbers 0 through 9, this program prints all odd numbers"*, 2) as part of a one-sentence summary preceded by a more detailed explanation, such as *"[the program] keeps adding one, starting at zero. If the number is divisible by two it will continue (skip) the number, if it is not it will print it. Basically, it will print all uneven numbers"*, or 3) as part of a line-by-line description: *"a variable i is set to zero. If the value of this variable is below 10 then the lines below will be executed. First the value of i will be topped up with one. Next, the program checks whether this new value for i, is an even number. If it is, nothing will happen. If the number is uneven, the value of i will be printed"*.

Other students gave no explicit indication of recognizing the importance of odd numbers to this program. Instead, these students mentioned that the program prints `i` (or the value of `i`) in a certain condition and provided a rather technical description of the code that is almost entirely line-by-line: *"this program takes the current number 'i' and adds 1 to it. Then it divides the number by 2, and if the remainder of the division is 0, it skips the value and starts at the beginning of the while loop with a new value i. If the remainder of the division is not 0, it will print the number 'i'"*. Sometimes these explanations were brief and contained (extreme) jargon: *"the start value is 0. In the while loop it will be increased with 1, if i modular 2 is equal to 0, i will not be printed otherwise i will be printed"* and *"We start at 0. while the index is below 10. increase i. return the remainder after division. continue: skips one element. print i"*. While it looks as if these students know how to explain the code well, no one-sentence summary was provided, nor can anything be concluded about their interpretation of the codes purpose (printing odds only). It is possible these students understood "what does this code do" rather literally in terms of the code's procedure.

Furthermore interesting to mention is that about 12% of the explanations reported the program *"prints not [certain values]"*, mostly without stating what the program does print. Finally, there are explanations that present a very vague description of the output. These include "prints the number/value" (28%) or even "prints a number/answer" (5%) without further specification of the number. This perhaps hints at little understanding of the program.

RPS-game (Figure 2.2d)

Contrary to the previous three exercises, the RPS-game provided different results regarding the focus of the explanations. Only one in three students (33%) mentioned any output generated by the print-statements in the program. Instead, a one-sentence summary was more commonly included (60%). Moreover, procedural information regarding the game, that corresponds to the different function definitions inside the program, was often included: setting a maximum end score (52%), entering a color (68%), the computer choosing a color (60%), determining the winner of a round (52%) and determining the winner of the game (42%). Based on the focus of the explanations, four types of student answers were identified (further addressed to in section 2.4.2) that mention:

A) a non-specified (N=3) or RPS-game (N=9), without explanation of the functions.

B) a RPS-game, (some) functions are explained (N=15).

C) a game (not specified), (some) functions are explained (N=16).

D) only explanations of (some) functions (N=23).

2.4.2 Presence and absence of elements

For our second research question, concerning the elements of a program that are most present or absent in students' explanations of that program, we looked to more detailed elements than just the core focus of explanations. These details include specific lines of code and specific code concepts. It is our assumption that the presence or absence of these details can provide insight in what the students deem important in their explanations and may reveal how students comprehend the code themselves. It was found that assign-

statements at the start of the code are often neglected in the explanations. It is possible that these statements are simply forgotten or altogether considered irrelevant to their explanation. In contrast, control-flow statements, such as if-else conditions and the start or ending of a loop, are often included. When function definitions are involved, they too are often included. It is likely that these elements contribute most to the comprehension of the program, as in the current case-studies, they were also at the heart of the programs. However, a difference can be seen between explanations with and without a one-sentence summary included in the answer. When such a summary is not included, students tend to focus on technical elements such as the increase of an index, a continue statement and user input.

If-else

Corresponding with the focus of the explanations being on the different possible outputs generated by the program, most explanations includes a form of if-else (93%). One in 5 students (N=34, 19.4%) (also) included the program has *"conditions"* and/or *"checks"* or *"tests"* the grade. However, the first line of the code (the input-function) was not always included. About one in three students (36%) neglected to mention that the user is required to give an input for this program to work. When it was mentioned, students showed very different ways of describing the concept. Examples are: *"[the computer / python / the program] asks the user to [type / fill / enter] a grade"*, *"the grade will be made by what you type yourself"*, *"the grade given by the user"*, *"the program allows you to put in a test grade of some sort"*, *"the program [accepts / takes] a number (as input)"*. Additionally, only one in four students (23%) explicitly mentioned the int()-function in their explanations. However, as will be discussed in 4.3, this function was often misinterpreted by the students.

For-loop

Regarding the for-loop, one interesting element is the printing of new lines. About sixteen percent (N=27) of the explanations contain an implicit or explicit mention of the print statements being separated by new lines. Half of these explanations show the exact output on separate lines either with or without further explanation. The other half mentioned explicitly that the output is printed on new or separate lines. Closer investigation revealed that there seems to be no pattern in how well the students comprehended the for-loop. Both implicit and explicit mentioning of the new line covered both correct and incorrect answers, such as *"prints hello 5 times"* or *"prints 1 2 3 4 5"* (see also section 2.4.3). One in five students that include a correct exact output (on new lines) also include further explanation of the code, for example explaining the range or how the loop ends (N=6, 22%).

While-loop

The while-loop offered good insights in what lines of the code are central to the explanations and therefore perhaps to the comprehension of the program. **Table 2.5** shows the presence of each element in the code snippet. Overall, the most mentioned elements are the range of the iteration (0-9; 84%), and the if-condition that checks for equal numbers (68%). The element that is most omitted from the explanations is the first line of the code, which sets the variable.

Table 2.5: Elements of the while loop with the number of times they were included in the students' explanations. A division is made between students who (also) included a one-sentence summary and those who did not.

Element	N (=110)	With one-sentence summary (N=34)	Without (N=76)
Set variable / start with 0	37 (34%)	3 (9%)	34 (44%)
Repeat / iteration / end of loop	92 (84%)	26 (76%)	66 (87%)
Increase i+1	55 (50%)	5 (15%)	50 (66%)
Mentions "modulo" or "remainder"	55 (50%)	8 (24%)	47 (62%)
… *of which mention "remainder" only*	41 (37%)	7 (21%)	34 (44%)
… *of which mention "modulo" only*	12 (11%)	1 (3%)	11 (14%)
… *of which mention both*	2 (2%)	-	2 (3%)
Checks if even / dividable by 2	75 (68%)	24 (71%)	51 (67%)
Continue / skip	59 (54%)	10 (29%)	49 (64%)
Print	103 (94%)	31 (91%)	72 (95%)

There exists a difference in distribution of the elements that are included or excluded between those explanations that contain a one-sentence summary, and those who do not. The biggest difference is seen with the first line in the loop (increase i+1), as two thirds of the explanations without such summary include a reference to this line of code, whereas only 15% of explanations that do include a summary refer to this line. A similar effect is seen with a specific referral to the modulo or remainder of the modulo, the continue statement and the starting variable. If we take into account that students who are able to abstract a summary or purpose of a program are usually considered to have better programming skills in general [Corney et al., 2014, Lister et al., 2006, Venables et al., 2009], this effect may not be surprising. Nevertheless, our findings confirm that without including a one-sentence-summary, students tend to focus on various specific and rather technical elements of the program to explain it.

RPS-game

Since this program was considerably larger than the other case-studies, it contained many different elements to present or omit in the explanation. As already shown in section 2.4.1, four groups were identified: A) (RPS) game without further explanation, B) RPS-game with further explanation, C) other game with further explanation, D) only explanation with no mention of a game. Apart from the distinctions mentioned before, other differences can be observed that are related to the elements of the code that are (not) presented by these groups (see **Table 2.6**). The most noteworthy observations are mentioned below.

Groups C and D almost always include a phrase referring to "user enters a color" (>90%). Most explanations in these groups start with this sentence. Moreover, these explanations are likely to include statements about the function checking the validity of the colors. However, in group C only one in five (17%) explanations mention an explicit ranking of the colors, compared to 40% and 35% in group B and D. Group C also seems to omit the function that specifies the maximum score most often and is least likely to include an output in words. Instead, explanations belonging to group C are much likelier

Table 2.6: Frequency distribution of different elements over four groups: A) (RPS) game without further explanation, B) RPS-game with further explanation, C) other game with further explanation, D) only explanation, no mention of a game. The percentages are of the total N per group.

TOTAL N	Total 66	A 12	B 15	C 16	D 23	Total %	A %	B %	C %	D %
RPS-game	24	9	15	0	0	36.4%	75%	100%	0%	0%
other game	19	3	0	16	0	28.8%	25%	0%	100%	0%
with colors	17	5	9	3	0	25.8%	42%	60%	19%	0%
play against pc	20	5	3	11	1	30.3%	42%	20%	69%	4%
one sentence summary	39	12	11	13	3	59.1%	100%	73%	81%	13%
exact output	5	0	0	1	4	7.6%	0%	0%	6%	17%
output in words	22	0	6	4	12	33.3%	0%	40%	25%	52%
mentions functions are used	7	1	1	2	3	10.6%	8%	7%	13%	13%
describes functions in detail	3	0	1	1	1	4.5%	0%	7%	6%	4%
enter a max points	34	0	9	8	17	51.5%	0%	60%	50%	74%
enter a color (user)	45	0	9	15	21	68.2%	0%	60%	94%	91%
—pc chooses random	39	0	11	12	16	59.1%	0%	73%	75%	70%
check validity of color	15	0	1	6	8	22.7%	0%	7%	38%	35%
—enter new color if not valid	11	0	1	4	6	16.7%	0%	7%	25%	26%
—prints something	7	0	0	2	5	10.6%	0%	0%	13%	22%
compare colors (calculate)	10	0	2	3	5	15.2%	0%	13%	19%	22%
—explicit ranking of colors	17	0	6	3	8	25.8%	0%	40%	19%	35%
—implicit “colors have hierarchy”	8	0	3	3	2	12.1%	0%	20%	19%	9%
—what happens when “tie”	13	0	3	5	5	19.7%	0%	20%	31%	22%
—determine winner (round)	34	0	6	13	15	51.5%	0%	40%	81%	65%
—allocate points	21	0	4	9	8	31.8%	0%	27%	56%	35%
—prints winner	7	0	1	1	5	10.6%	0%	7%	6%	22%
main function	7	0	1	3	3	10.6%	0%	7%	19%	13%
—connects previous functions	4	0	1	1	2	6.1%	0%	7%	6%	9%
—explicitly “if max_score reached”	13	0	3	4	6	19.7%	0%	20%	25%	26%
—repeat game (while loop)	18	0	7	3	8	27.3%	0%	47%	19%	35%
—game ending (winner)	28	0	9	9	10	42.4%	0%	60%	56%	43%
—prints the scores at end game	17	0	5	4	8	25.8%	0%	33%	25%	35%

to include that a winner is determined at the end of a round than explanations from the other two groups (C: 81%. B: 40% D: 65%). Furthermore, group B most often includes a reference to the program repeating itself caused by the while loop in the main-function, and, related to it, they also most often mention the end of the game (when a winner is found based on the maximum score). Finally, group D most often includes that a maximum score is entered (74%), usually right after or in the same sentence as the reference to “enter a color”.

2.4.3 Mistakes

For our investigation into the types of mistakes that students express in their explanations, only the first three assignments were analysed. Most strikingly, students showed various misinterpretations of programming concepts even after they had practiced them and used them by themselves in multiple assignments already, sometimes even for several weeks. Some of these mistakes are caused by a misunderstanding of programming concepts,

something that was especially visible with the for-loop explanations. Other mistakes may be caused by lazy or inaccurate reading, partly due to prior domain knowledge, or by an inability to explain the code clearly in words.

If-else

The concept of conditions and the if-else structure was well understood by the students. When misconceptions occurred they mainly concern the first line of the code: the input()-statement containing the int()-function. As mentioned above, the int()-function was mentioned only by one in four students (N=41) and often seemed to confuse or distract them; only eleven students that refer to the int()-function (36%) mention that the given input will be converted (or rounded down) to an integer. All others showed misconceptions. For example, multiple students mention that the input must be entered as a whole number or integer for the program to work. Other students only mention that the grade (or input) is a (whole) number. Occasionally students describe that the program will choose a random number. This last misconception probably shows a misunderstanding regarding input().

Next to misconceptions on programming concepts another type of mistake was interesting, and concerns the application of domain knowledge. In the if-else exercise the students could apply knowledge of the Dutch grading system (scale: 1 to 10; 10 being perfect, 6 being sufficient). Our analysis showed that knowledge of this system impacts the students' explanations in two ways: 1) they use it to explain the print statements, or 2) they use it to interpret (read) the code. For example, the code itself does not assume that it is impossible to obtain a grade higher than 10, yet multiple students have given this exact explanation to why the code prints "you are cheating" if the grade is any number higher than 10. Typical examples of these are: *"This code will grade your test. If you get more than the maximum grade the code will recognize it as cheating. (...)"*, and *"This [program] means that if you get higher than a 10, which is impossible, you are cheating. (...)"*

Regarding the use of domain knowledge to try and interpret the code, something else happens as well. A close read of the code tells us that a grade of 6 would print "try again". Contrary to the Dutch grading system where a grade of 6 is regarded as "sufficient", the print message in this code thus implies that a grade of six is not good enough. Using just their knowledge of the Dutch grading system, rather than closely reading the code, would therefore result in a mistake. Our results confirm that this is also happens: some students' explanations include that a *"six or higher"* is "sufficient" and/or prints "well done", whereas anything *"lower than a six"* would be "insufficient" and therefore print "try again". An important note is that students do not always include what should happen when the grade is exactly six, as they do mention *"higher than a six"* and *"lower than a six"*. These descriptions almost exclusively occur in explanations that explicitly show knowledge of the Dutch grading system, such as in the explanations mentioned above. Other explanations tend to use more explicit phrasings, such as *"higher than a six"* or *"between 6-10"* for "well done", combined with *"six and lower"* or *"everything else/lower"* for "try again".

These results may indicate that prior domain knowledge not only helps the students in understanding or explaining the code correctly but also contributes to wrong assumptions

about the code. In these cases, students may have gotten "lazy" in reading the code properly by thinking they already know what the code does.

For-loop

Even though the students had been practicing with for-loops for a whole week, the first line of this program (`print('hello'))`) proved to distract or confuse the students in the interpretation of the loop, revealing underlying misconceptions or poor comprehension of the construct. More than a fifth of the explanations (21%) propose that the word "Hello!" is printed five times. An example of this is: *"The program is asked to print the word Hello 5 times in a row"*. Of the 36 students making this mistake, 6 students combined it with the first print-statement, so "Hello!" will be printed six times in total. Some of these students explained their reasoning: *"Here the code is going to print "Hello!" 5 times. If you change the range it's going to print "Hello!" with that amount"* and *"First Hello! is printed/shown in your terminal. After that Hello becomes a variable in 'i', with range it is selected that this variable will move step by step 5 times. i is printed 5 times, or in other words: Hello! is printed 6 times in total"*. These students' explanations clearly shows a very fragile understanding of the for-loop, grasping the main idea, but making wrong assumptions about it's implementation.

The first line of the program also seems to mislead some of the students in a different way, letting them focus on the range of "Hello!" within the loop. Explanations of these students conclude for example that the program will print each individual letter on a new line (N=3), or that it prints the exclamation mark ("!") only (N=2). It is interesting to see that this kind of mistake was not isolated but was repeated multiple times by different students, and that they seem confident in their explanations: *"This program presents user with hello message. Then proceeds to present a rule in which the code traces a range of 5 and continues to print it. Resulting in the letters separated line by line"*; *"The program first prints Hello! After this it prints the sixth character: ! "* and *"the program will generate the word 'hello'. if there is an i in the word range, the program will print i"*.

Another difficulty in this exercise is visible with range(5) , which in Python counter-intuitively starts counting from 0, and stops before reaching 5. About half of the explanations (54%) include a correct reference to this, mentioning either zero to four or zero to five with an explicit explanation that five itself is excluded. However, thirteen explanations (8%) mention the program prints "0 to 5"; "0 – 5" or "0 till 5" while not providing extra explanation. This makes it difficult to read from the explanation whether the students have understood what will be printed. At least one of these explanations shows evidence that the student may have understood the concept of the for-loop rather well, while still making a little mistake in implementing the range-function: *"[…] print 0 then run the program again and print 1 etc till it prints 5"*. Other explanations (5%) show incorrect ranges too, mostly *"prints numbers 1 to 5"* or *"print 1 2 3 4 5 in separate lines"*. Sometimes this is combined with a further (correct) explanation of the for-loop: *"At first you print out "Hello", after that there is a for loop created which means the computer goes over the lines in the for loop as many times as given, in this case the computer goes 5 times of the for loop, because of the range(5) that was added. So in this case the output would be Hello 1 2 3 4 5"*.

While-loop

There are a couple of misconceptions or learning difficulties visible in the students' explanations of this program. About one third of the explanations showed signs of a misconception (32%, N=35). Half of them showed a difficulty with the continue statement (N=18, 16% of total) and a third showed difficulty with the modulo (N=11, 10% of total). Fourteen students showed (also) other errors, such as *"prints all numbers between 0-10"* or *"prints the remainder"*. Errors occurred with and without other errors.

Of the 18 students exhibiting difficulties with the continue statement, most of them (N=12, 11%) described their explanations in such a way that the program would print only even numbers instead of odd numbers. Two of these mentioned *"even numbers"* explicitly and three answers included exact output (*"2, 4, 6, 8, 10"*), either combined with a line-by-line description, a one-sentence summary or on its own. The other eight explanations contained sentences, including one-sentence summaries, like: *"if i can be divided by 2, i will be printed"* or *"this program will print i if it, divided by 2, has no remainder"*. This finding could suggest a misinterpretation of the continue statement. However, even students that show the ability of abstraction in one-sentence summaries are not spared from this mistake, therefore, we may also argue that this mistake can be due to neglecting the continue statement or perhaps lazy reading.

Other mistakes with the continue statement include 'vague' descriptions such as *"If the remainder is found after division is equal to zero, it is skipped. After that, i is printed"*, as well as (technical) descriptions that show no signs of understanding, like *"An object has value 0. If the object is under 10, execute a certain task"*.

Students that showed difficulties with the modulo-operator most often described the modulo as *"if the division is not equal to 0"*, *"if i/2 equals zero"* or *"the i that is equal to 0"* rather than mentioning the remainder of the division. This may be a direct result of the students either being unfamiliar with the concept, or not knowing how to describe it properly when they actually mean to say the remainder of the division.

2.5 Discussion

The aim of this chapter is to investigate what novice students include in their explanations when we ask them to explain code segments in plain English. We approached this by analysing student answers on given EiPE-questions that were part of a 12-week CS1 Python programming course. Special attention was given to what can be seen as the core focus of the explanations, which specific elements or lines of code are present or absent from the explanations, and what mistakes students demonstrate. Four case-studies were explored through open-ended, inductive and deductive coding.

2.5.1 What do students focus on?

It was found that the focus of the explanations, in the first three case-studies, was on the program's output as generated by the print-statements. However, the larger, more complex RPS-game showed a different pattern. In the explanations from the RPS-game we observed a smaller presence of the generated output and a larger presence of one-sentence summaries, input-statements and individual function-definitions. It is likely that this is

the effect of the nature and complexity of the code. Presumably, print statements are the first thing students look for when reading the code; this could be confirmed by follow-up think-aloud research.

Both print and input-statements can, and usually do, contain natural language. They can therefore aid the student's comprehension of the code when they recognize them as clues. This is especially true when such statements include information that link the code's structure to its purpose, for instance when the statement contains information about the programs context. One example of this in the analyzed RPS-game is a print statement that includes "the computer wins!", giving away that the program is likely to be a game against the computer. In the case of the analysed if-else program, the print statement "you are cheating" connects the condition to a natural language interpretation. Therefore, print and input-statements seem to serve as a kind of "translator" between technical structure and social purpose, aiding students in interpreting the code as follows from the Block Model [Schulte, 2008]. Moreover, these parts of the code can guide the reader towards a more focused reading strategy, as they may give indication of where to look next for important information. Although the students' reflections on their reading strategies were not analyzed as part of this chapter (see section 2.3.1 & 2.3.2), a quick run through their reflections seemed to confirm this hypothesis. Further research into students' self-reflections of their reading strategies may yield additional insights on this subject.

2.5.2 What do students include or exclude?

When looking at the more specific elements, or lines of code, that were included or omitted in the students' explanations, we found that some elements were almost always included, whereas other elements seem to escape students' attention in explaining the code. Besides print-statements and generated output, control-flow elements such as conditions for if-else statements, and the start and ending of a loop are most present. Another common pattern that was seen across the case-studies is that variable assignments at the beginning of the code is often omitted from the explanations.

However, and perhaps not surprisingly, the more elements there are in the presented code, the more variation we see in the descriptions. For example, the while-loop exercise, being complex enough to consist of enough different elements to choose from, while not being too big of an exercise to be overwhelming, showed that differences occur between explanations that include a one-sentence summary and those who exclude it. This is partly due to the fact that those who present a one-sentence summary do not always include any further explanation. However, it remains interesting to see which elements students choose to represent in that one-sentence summary. Previous studies have referred to such elements as possible "beacons" [Brooks, 1983, Pelchen and Lister, 2019] or primary goals of the program [Weeda et al., 2020].

More technical elements, such as increasing the index, a continue statement and the specific mentioning of the modulo or remainder, could be considered as secondary goals to the program. Perhaps students wish to be thorough and therefore include all elements in their description, but we could also argue that the students need these technical elements to explain the program. This would be in line with previous research which argues that relational answers (i.e. providing a summary of what the code does in terms

of the purpose of the code [Corney et al., 2014]) are related to higher scores on writing assignments and exams that test multiple programming skills [Corney et al., 2014, Murphy et al., 2012a, Sheard et al., 2014].

2.5.3 What mistakes do students make?

Pelchen and Lister [Pelchen and Lister, 2019] found a Java-code EiPE-exercise that proved strikingly difficult for students that only included the primary goals or beacons of the code. It is plausible that these students are only selectively reading the code, skipping, and therefore guessing, the parts that they do not understand [Pelchen and Lister, 2019]. In fact, some of the mistakes that the students demonstrated in the while-loop assignment in our current chapter can also underpin this theory. For example, we have seen students providing one-sentence summaries with wrong conclusions (*"this program will print i if it, divided by 2, has no remainder"*), which could mean that they misinterpreted the continue statement, or disregarded it completely, guessing the answer. In case of the RPS-game, we have seen three students just mentioning that the code represents a game of some sort, without recognizing the rock-paper-scissors structure to it. Furthermore, it is possible that the students recognized the RPS structure from just reading the input/print-statements and the variable names. If so, they did not need to read or understand the specifics of the program to explain its purpose. It would be interesting to test the students' comprehension from reading with a similar exercise, but instead with unfamiliar or meaningless variable- and function names and without revealing interpretations in the print-statements.

Another mistake, the misinterpretation of the for-loop, could reveal possible flaws in the instruction or the set-up of the course. Since the students were practicing with for-loops for a week, even making their own little programs with the concept, it was not expected that one in five students made the same major error in thinking that the program would print "Hello!" five times. The first print statement in this program clearly acted as a distractor for the students, who may have thought that the whole program was part of the loop. This line of reasoning could be the result of earlier exercises, in which students had merely practiced with stand-alone for-loops. Therefore, the students may yet have been unable to transfer their knowledge to a different context, reinforcing the notion that transfer of programming knowledge is not easy [Morrison et al., 2015].

Finally, our results on the if-else case-study indicated that prior domain knowledge was most often used to explain the behavior of this program. However, next to guiding students in their interpretation of the program, it also mislead them towards wrong explanations. Our results show that knowledge of the Dutch grading system interfered with a careful reading of the code: students stopped reading and assumed it followed the Dutch system instead. Future research could further investigate the effect of variable names in explanations. Prior work has already indicated that students who are better in explaining code are more likely to explicitly refer to variables [Pelchen and Lister, 2019].

2.5.4 Relation to other EiPE-exercises

Previous research has already shown the potential of EiPE-exercises when it comes to evaluating students' programming skills. As Lister [Lister, 2020] nicely summarizes it: *"the ability to answer plain English questions is a proxy; an estimate of a novice's ability to*

reason about code in an abstract way". Unlike prior work, where only the purpose of a code is central to the explanation, this chapter focused on what (else) students include, or exclude. It is the authors' presumption that in-depth knowledge of students' explanations after reading code may reveal learning processes, struggles and misconceptions that could be concealed when these explanations are primarily analyzed based on performance criteria (i.e. correct, complete, level of abstraction). Since the investigated assignments here are not specifically focusing on the purpose of a given code, our results may not be one-on-one extendable to classical EiPE tasks. However, our results do expand our knowledge on what students focus on while explaining a code, which is a crucial step also in translation to a code's purpose. In fact, even when specifically requesting the purpose of a code, part of the students seem to neglect it anyways from their answers. The work by Murphy et al. [Murphy et al., 2012a] shows that even when an extra prompt is given, so to only state what the code does overall, only 50-80% of the students returned a relational answer concerning the purpose of the code. Also work by Pelchen and Lister [Pelchen and Lister, 2019], who only include relational answers in their analysis, shows that many students do not include a relational answer when asked "explain in plain English what this code does": only 144 out of 344 students included four or more relational answers out of twelve EiPE tasks, which corresponds to roughly 43%. This clearly indicates that, although relational answers may correlate with better overall programming performance, explaining a given code is not easy for students.

2.5.5 Limitations

A threat to validity is the very generic question that the students answered to in this research ('explain what does this code do'). It is very plausible that different students interpreted the question in different ways, especially because they had no prior training in explaining code. Some students may intuitively aim for a general purpose, whereas other students, perhaps including students with autism spectrum disorder, may have interpreted the exercise rather literally. For example, they may be expecting that they are asked to describe the code line by line (focusing on the syntax), interpret "what does this code do" as "what does this code produce?" (focusing on output), or interpret it as "how does the program work?" (focusing on the code's execution). Furthermore, if a student chose to only include a general purpose, it does not necessarily mean that he/she ignored other aspects of the code, but could have deemed those irrelevant to their answer. Respectively, this impacts the extent to which student explanations can reveal their thinking or even understanding. Future EiPE research may consider using a less generic or clearer EiPE question that cannot be interpreted in multiple ways.

Since all explanations students provided were regarded as "correct answers" it is not expected that the exercises consistently increased students learning: no "model-answer" was provided before nor after the exercise and no feedback on their answers was given. Therefore, in theory, students could continue the curriculum without learning anything from the exercise. After all, the tasks were primarily designed to get the student more familiar with reading code, practice their reading skills and reflect on their understanding from reading. This of course interferes with the classic purpose of EiPE-tasks. Their idea is to learn to comprehend and explain code in a relational way, much according to the Block Model, encouraging the student to create a bridge between the structure of the code

and the purpose of the code.

A further limitation is that no comparison was made between the students answers and their overall grade on the course. Such comparison could greatly increase our understanding of students' learning success as it can differentiate between stronger and weaker performing students. Furthermore, since the RPS-exercise was given much further in the course, a straightforward comparison with the first exercises is difficult to make: students may have already established a stronger foundation of the programming concepts that were asked, or gained insights in how best to explain code. However, following individual students' performance was not part of this research. Finally, the group of students participating in the investigated course was not homogeneous, covering BSc Computer Science students, Honours Students (with different science backgrounds), and interested individuals. The different (programming) backgrounds of these students are likely to have somewhat influenced the results of this work.

2.6 Conclusions

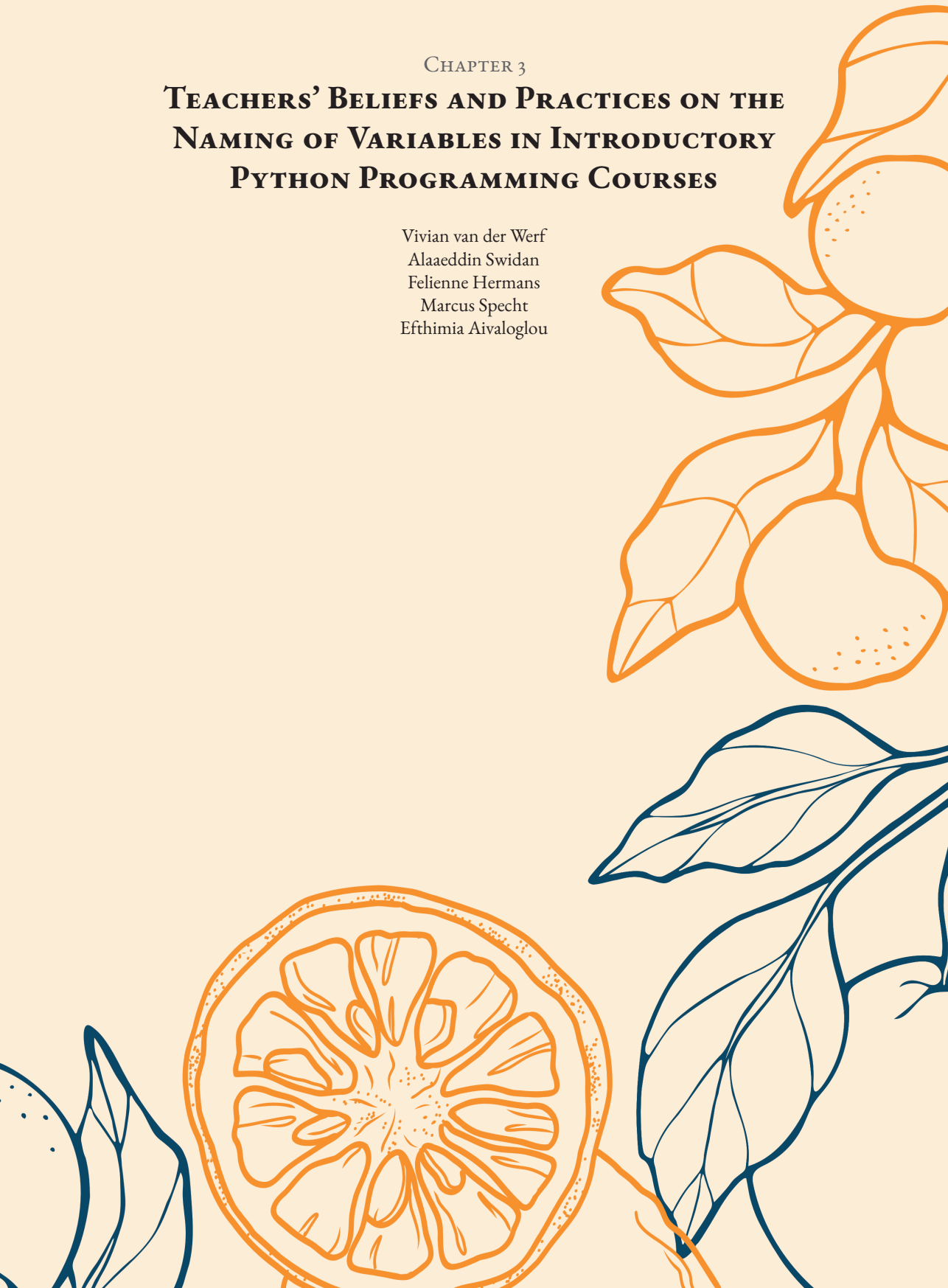
Our chapter investigated novice students' code explanations that were extracted from a university-level beginner course on Python programming. The results from our qualitative analysis show that student explanations may reveal specific struggles, possible flaws in the instruction, as well as elements of code that are especially important to students or that serve as a distractor. Such information can be used by teachers to improve their instruction, but also by students themselves when they are explaining their code to each other or have to interpret other people's code.

With this chapter, it is our intention to contribute to knowledge on the use of reading exercises in programming education and to inspire further research into EiPE-exercises as a possible instruction instrument. Related areas that we wish to explore in the future which are mentioned in the discussion are prior domain knowledge and students' self-reflections on their reading strategies. Additionally, we wish to further abstract patterns from the explanations into specific student (or explanation) types, look at the relation of these types with overall performance and programming skills, and explore individual progress of students once they practice more with reading exercises.

Chapter 3

Teachers' Beliefs and Practices on the Naming of Variables in Introductory Python Programming Courses

Vivian van der Werf
Alaaeddin Swidan
Felienne Hermans
Marcus Specht
Efthimia Aivaloglou



Abstract

Variable naming practices are part of the software developer's profession, influencing program comprehension and code quality. Yet, little is known about how variable naming practices are taught in beginner courses. This chapter investigates naming beliefs, self-reported teaching practices, and observations regarding variable naming practices of teachers of introductory Python programming courses. We adopted an in-depth qualitative approach by interviewing ten teachers from secondary education and higher education and developed several themes in order to answer our research questions. Among various opinions and practices, we found that teachers agree on using meaningful names, but have conflicting beliefs about what is meaningful. Moreover, the described teaching practices do not always match teacher's views on meaningful names, and teachers rarely encourage students to use them. Instead, they express that naming practices should not be enforced and that students will develop them by example. Whereas some teachers report focusing solely on conventions, others deliberately dedicate time for students to engage with naming, create their own guidelines, provide continuous feedback, or include naming exercises on exams. Naming practices do not seem to be deliberately taught, even though they influence program understanding and code quality. We also identified inconsistencies in teachers' self-reported naming practices. As such, we encourage intentional conversations about naming practices in educational settings, specifically linking naming to code quality and readability. We see room for group and peer activities as a means to this end, as well as providing formative feedback dedicated to naming. [1]

Keywords

variable naming
programming education
novices
teachers
qualitative interviews
open coding

[1] Published as: van der Werf, V., A. Swidan, F. Hermans, M. Specht, and E. Aivaloglou (2024). *Teachers' Beliefs and Practices on the Naming of Variables in Introductory Python Programming Courses*. In 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSESEET), **ICSE-SEET 2024**, page 368–379, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3639474.3640069

3.1 Introduction

Professional developers spend a significant percentage of their time (58%) on program comprehension-related tasks [Xia et al., 2018]. One root cause of this is that code is often written with 'meaningless' identifiers or variable names that are unintentionally misleading [Feitelson, 2023, Xia et al., 2018]. This causes problems in understanding and shows that finding a good name might not be straightforward. Accordingly, software engineering handbooks recommend professional developers to consider naming as a part of high-quality code, focusing on names' expressiveness, readability, and consistency [Stegeman et al., 2014, Börstler et al., 2017]. Evidently, naming plays a big role in understanding code [Avidan and Feitelson, 2017, Hofmeister et al., 2017, Lawrie et al., 2007b, Lawrie et al., 2006], which holds especially true for novice programmers [Teasley, 1994].

While some introductory programming courses include learning objectives that relate to code quality [Stegeman et al., 2014], several works already [Keuning et al., 2019, Börstler et al., 2017] noted that code quality, and naming in particular, do not seem to get equivalent attention in Computer Science Education research. Occasional efforts to incorporate naming include the development of code quality rubrics that involve naming as one explicit aspect to review or give feedback to students in their assignments [Stegeman et al., 2014, Stegeman et al., 2016, Glassman et al., 2015]. To the best of our knowledge, however, there is no research on teachers' perceptions of and approaches toward teaching (variable) naming in classrooms. We are interested in variable naming specifically, as variables are one of the first concepts taught in an introductory course, yet, the concept of variables is challenging for students to understand [Grover and Basu, 2017]. To this end, we conducted 10 in-depth interviews with teachers from secondary education, university, and adult education on the perceptions and practices of teaching the naming of variables. With these interviews, we aim to answer the following research questions:

RQ1 What are teachers' beliefs and perceptions about variable naming practices? We aim to identify how teachers think about names and naming practices in general, as we believe that these convictions are the background to which teachers adopt teaching strategies on the subject.

RQ2 How are variable naming practices taught? Here we investigate teachers' self-reported approaches regarding naming practices in the classroom. This question considers explicit (active) and implicit (or passive) teaching approaches, how teachers practice naming themselves, and information on feedback and grading.

RQ3 What do teachers observe in the classroom concerning variable naming? We are interested in what teachers observe in their students; for example, specific difficulties among their students and other observations.

3.2 Related work

3.2.1 What is good naming for comprehension?

Software engineering research indicates that programmers rely on names for their understanding of code [Avidan and Feitelson, 2017, Hofmeister et al., 2017, Teasley, 1994, Takang et al., 1996, Lawrie et al., 2007b, Lawrie et al., 2007a, Lawrie et al., 2006], and names often

serve as beacons during code comprehension [Gellenbeck and Cook, 1991]. Therefore, names can be of "poor" quality when they interfere with the reader's comprehension. In general, the following types of names are considered "poor" for code comprehension: names that are based on their data type (e.g. IntegerArray) or function within an algorithm (e.g. LoopCount) [Teasley, 1994], arbitrary names (e.g. GWhiz) [Teasley, 1994], and single-letter names [Lawrie et al., 2007b, Lawrie et al., 2006, Hofmeister et al., 2017]. Moreover, names can be unintentionally misleading and should therefore be chosen cautiously [Avidan and Feitelson, 2017, Arnaoudova et al., 2016, Fakhoury et al., 2020, Feitelson, 2023]. It was found that these misleading names resulted in more errors, taking more time, or giving up completely [Avidan and Feitelson, 2017]. Especially general, non-specific names such as "length" appeared problematic [Feitelson, 2023]. Research also identified multiple 'linguistic antipatterns' that misdirect a reader [Arnaoudova et al., 2016]. Common antipatterns that concern misleading names are names that "says one but contains many" or vice versa, and names that "suggests Boolean but type does not." Lexical inconsistencies like these significantly increase cognitive load [Fakhoury et al., 2020].

In contrast, there are also claims and observations on the effect of 'good' naming styles on code comprehension. Firstly, descriptive naming styles are advantageous over meaningless naming styles, like "Function1" or "FunctionA", even when documentation is provided [Blinman and Cockburn, 2005]. Additionally, meaningful abbreviations can be as effective as full-word names [Lawrie et al., 2006], and well-chosen abbreviations can in certain situations also be preferable over full words [Lawrie et al., 2007b]. When comparing letters, abbreviations, and full-word names, the latter still gives the best results on source code comprehension [Lawrie et al., 2006, Hofmeister et al., 2017], whereas letters *can* be meaningful if they convey information that is commonly attributed to that letter (i.e. "i" for index or "s" for string) [Beniamini et al., 2017]. However, attributions to specific letters vary over different programming languages [Beniamini et al., 2017], which of course has implications for learners of different languages.

Generally accepted recommendations on naming styles are that names must be picked with caution and given careful attention so that they reflect the concept or the role represented by each variable [Avidan and Feitelson, 2017], and, good naming consists of "limited, consistent, and regular vocabulary" with limited name lengths, so as not to overload a programmer's memory (the longer the name, the harder it is to retain the information) [Binkley et al., 2009]. Different roles of variables have been classified and investigated thoroughly in relation to comprehension by Sajaniemi and colleagues [Sajaniemi, 2002, Sajaniemi and Kuittinen, 2005, Sajaniemi and Prieto, 2005, Laakso et al., 2008].

On the subject of intermediate variables to break complex expressions into more manageable ones, Cates et al. [Cates et al., 2021] found that using an intermediate variable is generally beneficial for understanding only if the used name also reflects the meaning of the variables. Concerning camelCase and underscore styles, no difference in accuracy is found between the two styles [Sharif and Maletic, 2010]. Finally, naming styles are related to code quality [Stegeman et al., 2014]. For example, poor-quality identifiers (especially at the method/class level) are associated with lower quality, more bugs, and less readable source code, and natural language and recognized abbreviations can function as indicators for source code quality [Butler, 2009, Butler et al., 2010, Butler et al., 2009].

3.2.2 How do developers use names?

Naming of all identifiers, including variable names, accounts for over 70% of all characters in open-source projects and covers about a third of all tokens [Deissenboeck and Pizka, 2006]. Beniamini et al. [Beniamini et al., 2017] showed that single-letter variable names are common practice, quoting "in C, Java, and Perl they make up 9–20% of the names." Gresta et al. [Gresta et al., 2021] investigated Java naming practices in open-source projects and found that the three most common names are 'value', 'result', and 'name', while also single-letter names, like 'i', 'e', 's', and 'c', are commonly used. In twenty open-source systems that use the languages C, C++, C#, and/or Java, Newman et al. [Newman et al., 2020] looked for the most common grammar patterns in several types of identifier names and found that names typically have a singular noun-phrase grammar pattern (i.e. 'nextArea' or 'max_buffer_size'), with the exception of function names or when representing a Boolean value. More than three-quarters of identifiers containing a Boolean include a verb, likely to show that a question is answered by a true or false. Moreover, plural names often refer to a certain collection (of lists, arrays, etc.) or data grouping. Peruma et al. [Peruma et al., 2018] found that if identifiers are renamed, it is to narrow the meaning and serve code comprehension. Recently, Feitelson et al. [Feitelson et al., 2022] found that the probability that two developers choose the same name is very low, although different names are understood by the majority of developers. They suggest a model to help developers choose better names; in short, select the concepts that need to be included, choose words to represent those concepts, and construct the name with these words.

Swidan et al. [Swidan et al., 2017] analyzed projects in a block-based language originally directed at children (Scratch) to see how variable names are named there. They found that these names tend to be longer than in other languages, with most names between four and ten characters of length and only 4% of names being single letters. When single letters are used, 'i', 'x', and 'y' are the most common, showing both a crossover between languages and a focus on coordinates; the latter reflecting the focus on games and animations in Scratch projects. Additionally, Swidan et al. [Swidan et al., 2017] found that over half of the variable names are single words, with another 30% having a maximum of one space. This suggests that Scratch developers either use underscores or casing to separate words, like in most textual languages, or that single words are most naturally chosen by Scratch developers.

To support using consistent and concise names, a tool was developed striving to follow certain composition rules [Deissenboeck and Pizka, 2006]. This was followed up by Lawrie et al. [Lawrie et al., 2006], who then confirmed prior conclusions that programmers use a limited vocabulary [Antoniol et al., 2002, Caprile and Tonella, 1999]. Furthermore, Butler et al. [Butler et al., 2015] created a library checking naming conventions in Java, also in the context of using certain typography, abbreviations, and phrases. They found that about 85% of Java projects follow standard conventions. Allamanis et al. [Allamanis et al., 2014] presented a framework that learns the style of a codebase and suggests revisions to improve stylistic consistency. They noted that "almost one-quarter of the code reviews examined contained suggestions about naming," highlighting again the relevance of proper naming.

3.2.3 Naming in programming education

In comparison to experienced developers, proper naming styles (expressiveness, readability, and consistency) might be especially relevant for novices learning to program. For example, bugs are easier to find when words are used [Hofmeister et al., 2017], suggesting that good names improve code comprehension and debugging. Additionally, when developers use 'descriptive compound names' (i.e., "convertedInput" instead of "result"), they change their reading direction less often to find and correct a semantic bug, which they do 14% faster than when normal names were used [Schankin et al., 2018]. The effect, however, was stronger for experienced developers compared to novices, which suggests that novices do not benefit the same way, perhaps because they have not learned to 'interpret' specific naming customs yet. In fact, novice programmers often fail to name variables correctly [Gobil et al., 2009] and Scratch students are found to be misled by variables named with a letter, probably because of prior knowledge from their mathematics education [Grover and Basu, 2017]. These findings highlight that opening a discussion between teachers and students about "what is good naming" might be more relevant than just pointing toward naming conventions created by experts. This notion is strengthened by the work of Glassman et al. [Glassman et al., 2015], who, in the context of improving online curricula, developed a tool and a quiz for their MOOC to assess naming in terms of length and vagueness. As a by-product of their tool they found that feedback on naming practices, as well as both good *and* bad examples, was highly valued by students. Also Börstler et al. [Börstler et al., 2017] found that feedback related to code quality was frequently asked for by students. This suggests that topics such as readability, including naming, might not get enough dedicated attention in educational programs.

3.3 Methodology

This study aims to investigate teachers' beliefs, practices, and classroom observations on the naming of variables. Similarly to other works in CS education research [Keuning et al., 2019, Tshukudu et al., 2021], we captured such information by asking teachers directly through the means of semi-structured interviews.

3.3.1 Recruitment and teacher demographics

We targeted a wide range of teachers, including teachers at secondary school, university level, and adult education, who currently teach or recently taught one or more Python introductory courses. Teachers were recruited internationally through the networks of the authors and through the national network for secondary school informatics teachers. Teachers were required to speak either English or Dutch during the interview but could speak a different language in the classroom. Before the interview, teachers gave informed consent and completed a short questionnaire covering their backgrounds, such as programming experience, teaching experience, and other demographics. An overview of the recruited teachers can be found in **Table 3.1**. To minimize self-selection bias concerning naming specifically among volunteering teachers, they were approached with the topic of variables in general, not on the topic of variable naming.

In total, we conducted 12 interviews, with 7 teachers from 4 different universities, 4 teachers from 4 different secondary schools, and one teacher in professional "on-the-job"

Table 3.1: Overview of participants; formatted as: T1Um = [T]teacher [1], [U]niversity teacher, [m]ale. Teaching experience (programming and all) is counted in years. No. of students is per class.

ID	Education	m/f	Course target group	Age	Teaching exp. prog.	Teaching exp. all	No. of students
T1Um	university	m	CS + engineering BSc	25-34	2	2,5	70
T2Um	university	m	CS BSc	45-54	20	20	400
T3Am	adult	m	IT professionals	55-64	1	1	1-to-1
T4Sm	secondary	m	HAVO/VWO, optional	25-34	4	4	50-70
T5Um	university	m	CS + engineering BSc	55-64	16	17	400
T6Um	university	m	CS + engineering BSc	35-44	9	23	400
T7Sm	secondary	m	ICT track, mandatory	25-34	8	8	5-20
T8Sm	secondary	m	VMBO/HAVO, optional	25-34	2	5	25
T10Uf	university	f	CS BSc; CS MSc	25-34	1	1	300
T11Sf	secondary	f	HAVO/VWO, optional	<25	1	3	20-30

coaching. Participants worked in The Netherlands, Belgium and Spain. Their mother tongues were Catalan, Dutch (incl. Flemish), French, Italian, and Turkish. The participating university teachers taught in English or Dutch, whereas the secondary education teachers all taught in Dutch. Our participants' teaching experience ranged from 1-20 years, indicating we recruited both experienced as well as starting teachers. Most teachers program themselves, with the exception of one secondary school teacher. Eight teachers also taught other languages besides Python.

3.3.2 Interview process

We used a semi-structured interview protocol consisting of questions on three topics about the teaching of variables: (1) general perceptions, (2) teaching practice, and (3) student difficulties (see Appendix 3.6). To capture a broad view of teachers' perceptions, practice, and experiences, each topic contained various neutrally posed open-ended questions that offered room to dive into detail with follow-up questions. The interview covered both the concept of variables in general and the naming of variables specifically. Variable naming was covered in all three topics both as part of specific predefined questions and as follow-up questions during the interview. Each participant was given an equal opportunity to talk about naming. When naming did not come up naturally, the topic was introduced via follow-up questions. However, not every participant spent equal time on the topic: in cases where the interviewee was not able to elaborate any further, the interviewer moved on to other questions. A pilot interview was used to test and inform the interview protocol, after which it was decided that no further alterations or refinements were needed.

All interviews were conducted online, by the first author, via MS Teams, and recorded for transcription. The average length of the interviews was 62 minutes, about half of that time was dedicated to the topic of naming. The interviews were transcribed manually, in the original language of the interview (Dutch or English), using intelligent verbatim transcription. Transcripts were checked for discrepancies and made anonymous for subsequent processing. From the 12 interviews we conducted, two were excluded from the final analysis: one teacher did not teach Python programming despite indicating this

beforehand, and one interview was considered a duplicate as it was with an assistant who taught alongside a previously interviewed teacher and revealed no new information.

3.3.3 Coding process

To obtain a broad overview of variable naming practices in classrooms, we used a qualitative open-coding approach from a constructivist perspective [Corbin and Strauss, 2008, Kenny and Fourie, 2015, Charmaz, 2014].[2] This means that we prioritized information generated by the data in an intuitive way, rather than creating a framework or hypotheses based on literature which then is used for deductive coding. Accordingly, the complete transcripts were analyzed using an iterative process ("open coding" and "refocused coding"), which means that *all* quotes throughout the interview related to variable naming were coded in a way that best summarizes the quote's intent and meaning [Corbin and Strauss, 2008, Kenny and Fourie, 2015, Charmaz, 2014]. This process is known to generate a large set of individual (in-vivo) codes that can be grouped and merged into themes according to the research interests. In this case, the first author coded three interviews first and used the open codes from these interviews to develop a more structured codebook. The initial version of the codebook consisted of grouped themes that provided information about our research questions, such as "naming beliefs" (RQ1), "teaching strategies" (RQ2), "grading and feedback" (RQ2), "own representations" (RQ2), and "student observations" (RQ3). In iterative rounds, the first author, together with the last author, also identified preliminary main categories that gave direction into the specific topics that teachers brought up. These categories distinguished, for example, between various perspectives (i.e., focusing on meaningful names or letters), teaching strategies (i.e., active or passive), and teachers' own identification of their representation of naming (i.e., using letters or full words).

Using the developed categories as a guideline, the coder then (re)coded all 10 interviews, still maintaining a semi-open coding approach up until saturation was reached. This means we continued creating new open codes if needed, but mostly added codes and quotes to existing themes and categories. This process was done iteratively and repeated for already coded interviews when new insights were made. New insights also meant that the codebook was updated: new themes and categories were added, renamed, split, or merged until all relevant and remaining open codes were summarized and grouped into categories with similar meanings and intentions. For example, the old theme "teaching strategies" was split and renamed into "active" and "passive" teaching approaches, each with its own categories to more accurately describe and interpret the information given by the teachers. During this process, doubts were discussed with the last author during regular meetings, in which the last author also checked the emerging categories and themes for clarity and validity. The final codebook is presented in **Table 3.2**. In total, we ended up with 238 individual codes to analyze. The tools used during the data processing and analysis were Atlas.ti and MS Excel.

[2] Although this research follows Grounded Theory (GT) procedures, the intent of this work is to gather various existing perspectives and teaching practices among a variety of programming teachers. Since we know of no prior work attempting to create such an overview, we considered an iterative process as used in GT procedures most intuitive to discover patterns in teachers' own descriptions.

Table 3.2: Final code book with examples of codes per developed theme, category, and RQ. Examples of specific quotes (utterances) can be found in-text in the results section.

RQ	Theme	**Category** (# of codes)	**Examples of individual codes**
1	beliefs and perspectives	meaningful names (47)	describes what is in the variable; provides its function; should be intuitive
		using letters (16)	i, j, k, n, l are not very informative
		size and detail (14)	prefers longer names; depends on situation
		overall structure (30)	use a certain structure; use underscores
2	active teaching approach	coding conventions, rules, guidelines (7)	uses or focuses on a personal coding guideline; focus on community practice
		readability, programmers attitude (7)	focus on readability; focus on job expectations; focus on human aspect
		other active approaches (6)	focus on errors (pointing out, discussion); stresses code works independent of naming; provides explicit naming assignments
2	passive teaching approach	mentioning no teaching (4)	not explicitly taught; no discussion on naming
		students learn by practice (12)	naming comes naturally; lead the way; only during other assignments
		other (5)	no specific way is required, taught on demand
2	own representation	"meaningful" or similar term (8)	representative; descriptive
		single letters or abbreviations (5)	uses single letters: loops; uses single letters: basic operations
		depends (2)	depends on the program; depends on the purpose
		other (5)	practical reasons, no particular style
2	grading and feedback	no evaluation (4)	not graded; no points deducted
		unclear (5)	part of general assignments; part of other skills
		evaluated (6)	graded on test; continuous feedback
3	student observations	difficulties (23)	typos; too long names; what is 'i' in a loop
		causes of difficulties (10)	students lack creativity; confusion because of renaming
		other observations (22)	better students give better names

3.4 Results

3.4.1 RQ1: What beliefs do teachers have about variable names?

Below we present different topics that teachers mentioned when they reflected on naming practices. The results are summarized by statements reflecting teachers' beliefs about variable naming.

Names should be 'meaningful' and 'intention-revealing'

Most teachers agree that naming is important and should be meaningful. Names are considered meaningful when they are simple, straightforward, and intuitive. They have to be descriptive, clearly represent the contents of the variable, or show its purpose. Mentioned examples are usually nouns: *studentName*, *interest*, *length*, *result* or *index*. To sum up what is regarded as 'meaningful', T1Um tells his students: *"try to name it a name that makes sense to you and two other people."* He also notes that the addition of adjectives, for example, *current_maximum*, can be extremely helpful in loops, but should be used *only* when it makes sense, to avoid new confusion. For example, if there is only one maximum in the code, adding a *current* to *maximum* is not helpful. Moreover, names are to be intention-

revealing. Teachers emphasize this specifically when it concerns functions: names have to reflect the function's purpose so that *"just by looking at the name of the function you can tell, okay this function is supposed to perform this, and so on" (T11Uf)*. Mentioned examples are structured with a verb to indicate it "does something": *calculate_weight*, *organize_file* and *find_cost*.

Teachers disagree on using letters as names

Letters and abbreviations are generally considered to provide too little information to be descriptive. Especially in the context of teaching, T4Sm explains: *"if I start using very bad names like 'a', 'b', and 'c', then, the code still works the same way, but it's not telling students what it does. And it can be a good exercise, but not in the parts where I'm explaining what they do. It's a good exercise on a test where [the students] need to know better but not during teaching or not during comprehending the concept that I'm trying to explain."*

However, there is disagreement on whether letters should be used. In particular, T11Sf mentions that *"letters in the case of operations are meaningful because [my students] can easily relate it to their math classes from before, which makes it an appropriate naming scheme."* Also, T8Sm remarks, *"with small assignments I will use letters, especially with basic math operations, using 'a + b' is just more logical than writing 'number1'. It's more like mathematics" (translated)*. Another consideration to use letters is the traditional practice in the (online) community. This is especially true for (nested) loops, where the use of *i - j - k* is common practice: *"if [students] would google to something, they would find it like that. So I try to teach them also in how they would find it if they would google online" (T11Sf)*. However, some rather use an *x - y - z* structure in nested loops: *"Now, for me [i-j-k] is an example that it doesn't make much sense because if I'm going through a matrix in which there is an 'x' and there is a 'y', why am I using 'i' and 'j'? I know, it's tradition to use i-j-k etc., I just think that in some cases it would make more sense to use 'x' and 'y'. (...) Imagine that I want to use 'x in y', I have to put 'i in j', and then, depending on how long is the loop, I have to remember by heart that 'i' is 'x' and 'j' is 'y'" (T1Um)*.

Whether letters are considered meaningful thus seems to depend on whether the letter itself carries meaning. In other words, using random letters from the alphabet is generally viewed as 'bad practice' whereas particular letters are accepted in certain contexts, like loops, short codes, or codes that are not intended for sharing: *"If it is for myself and nobody else is ever going to see it, I might even use 'x', 'y', 'a', but as soon as it's something that I will share... yeah, no, for sure. I put the variables with the right names. I have to consider the fact that somebody else is going to read this" (T1Um)*.

Names have an ideal size and level of detail depending on context

Several teachers report that names should not be "too short," or "too long." As is the case with random letters, it is reasoned that names that are too short create confusion because they do not convey enough information to the reader, which in turn makes it hard to remember what contents are behind which names. Too long names, on the other hand, create confusion because the reader might not read the whole name and rather assumes its contents or function after reading only a part of the name (T4Sm). Teachers furthermore mention that "enough detail" should be provided, but not "too much." For example, the name *student* is not detailed enough when its contents are numbers: it remains unclear

if the variable represents a student's age or grade or something else. On the other hand, variables named *Max* with contents "Max" or *seven* with an integer 7 are "too detailed," as well as *all_names_of_name_list_starting_with_a*.

According to teachers, writing efficiency also affects the balance between longer and shorter names: longer names are less efficient and more prone to typing errors (T8Sm, T4Sm, T10Uf). However, T11Sf mentions, *"especially with beginners, I would prefer the longer names, where we give the purpose of the variable or what it does over short and concise names, even though I get that it's more time-consuming."* Nevertheless, the ideal size of a name varies per teacher and context. Short names, and even single letters, are considered "okay" in short programs, whereas in longer programs names should also be longer (T1Um). The simplicity and conciseness of a single word are valued, but only if the name is unambiguous in its meaning. A maximum of one to three words are preferred, connected with an underscore or via casing.

Overall structure is important but naming is a personal style

The overall naming structure and the relationship between names are considered important. Some (T1Um, T6Um, T10Uf) prefer a numbered structure, for example, *plant1, plant2, p1, p2, a1, a2, example1, example2, str1, str2, df1, df2*, or a logical structure between the names. However, T3AM cautions that such structures can get too complex and confusing, for example, when names are structured like *aa, ab, ba, bb*, etc. Moreover, T1Um and T4Sm stress that names should not be too similar to each other to avoid confusion between names (*apple* vs. *apples*). Additionally, T4Sm also warns that *"if all or most variables look the same, students think it should be done that way."* He experienced this with a structure consisting of *myInput, myText, myInt*, as used in a KhanAcademy module: students copy it, and start creating names such as *myLastTwoValues*. This *"does not help and is not mandatory (...) It is not bad, but it is not what I expect from [my students] when using variables (...) [and they] have to be able to make more complicated names if necessary."*

Naming conventions are also mentioned as important. While T11Sf prefers following traditional Python or community guidelines (i.e. PEP), others adopt self-created guidelines in their teaching (T4Sm, T7Sm). For T2Um and T6UM, using a certain naming style is not very important, as long as their students are consistent. Furthermore, depending on the teachers' own programming background they prefer underscores over camel-case or vice versa, for example, T1Um: *"I do like underscore because it gives me a visual interruption."*

Finally, some teachers consider names that include data types to be helpful to novices or prevent issues when (accidentally) combining data types. For example T10Uf, *"I try to associate the variable with its type. If it's a list then the name has a list, if it's a string then the name (...) most likely is going to have a string in its desirable name"* and T3Am, *"to keep a certain type-safety or reminder by including it in the name (...), especially for beginners, I recommend using naming that is as clear as possible, and possibly even include data types" (translated)*. However, since Python is a dynamically typed language, T7Sm notes: *"it is not that important for students that don't use that kind of programming languages to really be constantly reminded of the datatype"* and also T6Um mentions: *"in my opinion, it's not necessary. (...) I don't have a tendency to say that the type should be reflected in the variable name."*

3.4.2 RQ2: How are naming practices taught?

We found various teaching practices that we grouped into *active teaching approaches* - naming is explicitly taught or mentioned in the classroom, *passive teaching approaches* - naming is not or implicitly taught, *own representation* - the way teachers use naming themselves when teaching, and *feedback and grading* - whether or not naming is evaluated.

Active teaching approaches

We consider teaching approaches as active when naming is explicitly taught as part of the curriculum. There are two major topics: (1) coding conventions, including guidelines, community practice, and specific naming rules, and (2) readability, including clear and meaningful naming and the human aspect, such as teamwork and job expectations.

Coding conventions, guidelines and rules. Most teachers mention coding conventions; however, T11Sf remarks: *"I try to use the conventions of the languages, but that is quite difficult when the students learn multiple different languages during the three short years that they have computer science."* Consistent with this statement, conventions, guidelines and rules are not very homogeneously taught among the teachers, which complies with the diverse beliefs we have identified among teachers concerning naming practices. Some teachers set up their own naming guidelines or recommend students to make their own structure, others mention to include tradition and community practice (i.e. PEP) in their teaching and focus on naming conflicts or recommending their students to include data types in their names. T5Um mentions consistency to be most important in teaching naming: *"what I would stress is more that things are done in a consistent way rather than having, doing it always one way or another; the point is you shouldn't mix and match in the same program different styles, whether it's for naming variables or for even programming style or indentation and the comment style, all of that."* To help students develop their naming practices, our teachers regularly mention to provide tips and show examples. Additionally, tools such as Visual Studio Code or PyCharm are sometimes adopted for correcting and teaching coding guidelines.

Readability, meaningful naming, and the human aspect. Most teachers merely mention to students to use clear and meaningful names, for example: *"We do insist on trying to give names which are as readable and as complete as possible" (T2Um)*. However, some teachers (T4Sm, T6Um, T7Sm, T8Sm) (also) discuss why naming is important. This usually includes a human aspect such as organizing your code to remember or find stuff back. Other human aspects are working in teams, future job expectations, and naming something in a way that you and at least two other people can understand what you mean. When names are not "readable," or, "according to the set guidelines", T7Sm goes as far as telling his students *"Okay this thing, I don't know what you mean here so I can't read your code, right now"*, even if he does understand the names. Additionally, he likes to prepare the most frequently seen mistakes in student projects, in order to discuss and evaluate them in class and to show how students can improve their own projects. With these strategies, he wishes to provide continuous feedback, prioritize the importance of naming, and motivate his students to first fix naming issues before they can get help from him on other aspects of the code.

More active approaches. Teachers also indicate using strategies such as pointing out naming errors (T1Um, T7Sm, T11Sf), discussing mistakes in class (T7Sm), and providing explicit naming assignments (T4Sm, T7Sm) that include bad smells and error-hunts. T1Um explicitly stresses that code works independently of naming and that naming therefore is only important for a human reader: *"I put a lot of stress on the fact that [naming] does not matter but that it matters on our level of organization. (...) I don't oblige them to rename [their variables] because (...) I want to stress that the code could work anyway. (...) You could call it 'banana' and it works, you just have to know where to put 'banana'. On the other hand, I also tell them it has to make sense for somebody who reads it."*

Self-reflections. Several teachers reflected upon their own practice and mentioned wanting to incorporate more specifics about the practice of variable naming. T8Sm: *"This is something that I now will start to think about much more than I ever have before, that is also nice for me" (translated)*, and T4Sm: *"I underestimated how I teach variable names because I thought it was one lesson and involved less and I can teach them everything about variables [in one lesson]. But I've already split that into two, three lessons, just for Python. Not only because it's not as uncomplicated as I thought, but also because it's a lot bigger than I thought (...) It has to be done because it's not as natural as I think it worked."*

Passive teaching approaches

We consider passive teaching approaches all strategies that do not *explicitly* teach naming practices. This includes all statements where teachers mention that they do *not* give specific attention to naming practices, and all statements pertaining to practices where students are (sometimes explicitly) assumed to learn by themselves. This thus involves indirectly taught naming practices (i.e. "through general exercises" or by "leading the way"). Furthermore, some teachers do not require students to use specific naming styles. **Table 3.3** presents an overview.

Table 3.3: Overview of passive teaching approaches used

Passive teaching approach (N)	Teachers
No explicit focus on naming (6)	T2Um, T5Um, T6Um, T8Sm, T10Uf, T11Sf
Naming is practiced through examples (8)	T1Um, T2Um, T3Am, T4Sm, T5Um, T8Sm, T10Uf, T11Sf
Specific naming is not enforced (6)	T1Um, T2Um, T5Um, T8Sm, T10Uf, T11Uf

No explicit focus on naming. Teachers report having no specific focus on naming in their courses. For example, T2Um reports: *"we don't have an explicit theory session where we talk about the naming conventions for variables would be this or that"*, and T11Sf mentions: *"It is not something that I start focusing on but it is something that [students] do start noticing along the way."* Interestingly, this finding is in contrast to what we see under *active approaches*. Specifically, teachers tell us not to have a specific focus on naming practices, while they *also* indicate telling students to "choose meaningful names" or to

"follow the conventions." However, teachers with this inconsistency do not seem to follow up their instructions with further explanations or assignments; instead, they remain with general tips. When elaborating on why they do not explicitly teach naming practices (see below), teachers assume that students will pick up "good naming practices" on their own and that naming practices do not require more explicit teaching or attention. It is also mentioned that naming practices should not be enforced as it is seen as an individual choice or preference.

Naming is practiced through examples. Teachers assume that students learn naming by themselves, either by following the traditional or given conventions or through practicing in other assignments. For example, *"so not very explicitly, but often naming is featured in the context of an assignment [that shows] that it eases understanding" (T8Sm)*, and *"we introduce the rules as we go, by the examples we give them" (T2Um)*. Even more strongly, T1Um chooses to lead the way as he assumes his students will copy him: *"I do it passively. For example, saying, "for 'index' " because they're indexes, "for 'length' " because it's a list of lengths. So I try to make them get there."* Furthermore, T5Um argues that naming practices do not need explicit teaching. He states *"I don't insist on [naming] very much (...) I mean, that comes more naturally by example"*, and emphasizes that it is not *"a major source of concern for the students"* as they are confronted with a lot of code through exercises, examples, and their own written code. Following this, he mentions: *"I don't think naming is a big concern to us [teachers]."* Two more teachers mentioned focusing only on naming *if* and *when* that was necessary, for example, in the context of an error or when the topic was brought up by a student.

Specific naming is not enforced. Teachers do not like to emphasize -or require- specific ways of how variables should be named, but rather leave it up to the students. T5Um: *"we don't specifically insist very much on how variables should be named"*, and T1Um: *"I don't want to stress a lot they have to use these names or use that name."* Instead, T2Um tells his students: *"It's okay, your code will work and it is not so important in this course, we are happier if your code works."*

Own representation

We consider how teachers use names themselves when they are using examples or show live coding to their students as their *own representation* of naming practices. It can be seen as setting an example to the students, as such, we also consider teachers' own representation a passive teaching approach. However, since it is always present in a course, and therefore complementary to other approaches used, it deserves its own category.

Almost all teachers report that they use, or try to use as much as possible, meaningful names or equivalent terms. For example, T1Um: *"I'm very straightforward, so like for (...) doing the for-loop, I do: for index in the list of indexes. Because they're indexes, so, let's use index."* Interestingly, many equivalent terms are mentioned (see **Table 3.4**), possibly showing that no one single way of good naming is present, and perhaps also showing slight nuances in what teachers find most important in choosing a name. The variety in the self-reported own representations presented here is consistent with the variety of naming beliefs that we discussed previously.

Table 3.4: Descriptions used by teachers to show what type of naming they use themselves when teaching. Interpreted as variations of "meaningful names."

Description used	Teachers
meaningful names	T4Sm, T5Um, T11Sf
clear names	T3Am, T4Sm, T10Uf
representative names	T3Am, T4Sm, T8Sm
descriptive names	T4Sm, T8Sm
straightforward names	T1Um, T3Am
informative names	T6Um
adequate names	T5Um
useful names	T5Um
one-letter names	T5Um, T8Sm, T10Uf, T11Sf
abbreviations	T8Sm, T6Um
no one-letter names	T3Am, T6Um

Some teachers note that their naming depends on the program or the purpose of the code. While T3Am and T6Um explicitly told us that they do not use one-letter names, others told us that they do use abbreviations and one-letter names, sometimes depending on the purpose of the program or the complexity of the code. Single letters were especially used when teaching loops and basic (math) operations, for reasons already discussed in Section 3.4.1 (*Teachers disagree on using letters as names)*. In particular, these reasons concern a connection to prior knowledge (mathematics) and tradition or community practice. Moreover, short names, abbreviations, and single letters were also used for practical reasons or convenience. T8Sm: *"If it doesn't matter much, or the code is small, I usually use just a letter, to have overview [and] for time efficiency. If the code grows larger or more complex I prefer abbreviations" (translated)*. T6Um: *"I would prefer to avoid these too short names, although, actually, on some of my slides, I do use these short names."* His reasoning is to avoid using a font size that is too small while still being able to compare two pieces of code on the same slide. During the interview, he realized that *"at the same time, if you do that too often you give a bad example, that's... that's a difficulty [laughs]" (T6Um)*. Interestingly, as his first response to the question of how he used names himself in teaching, he said: *"I like consistency a lot (...) that you approach things in a consistent manner, that students get a certain, learn a certain way of thinking"* (T6Um).

Feedback and grading

The topic of feedback and grading came up in 7/10 interviews, from which we identify three approaches to evaluating naming practices: (1) no evaluation, (2) indirectly evaluated or plays a minor role in grading, and (3) explicit grading and/or feedback. First, there is a strong tendency to not grade or evaluate students on their naming practices. Most teachers explain that naming is not part of the evaluation of student's work, or that students do not get "punished" (i.e. subtraction of points) when improper names are included in their submissions, for example, T10Uf: *"We don't do a lot with variable naming (...) I don't think we pay attention to readability."* One reason mentioned is that auto-graders do not look at naming quality. However, teachers indicate that it does not make sense to grade it separately since naming is interwoven with performance on other concepts (T7Sm,

T10Uf). Even though consistency within a code is often desired, this is not enforced (T5Um, T8Sm).

Second, when naming is part of grading it is usually connected to "programming basics," "using conventions" and "good commenting," or it is graded through practice with weekly assignments. However, although these weekly assignments are not necessarily focused specifically on naming practices, they cover various programming topics, including variables, and they are explicitly mentioned by the teachers as opportunities to practice naming. Therefore, how naming is actually evaluated remains unclear.

Third, two teachers show explicit evaluation of naming practices. T7Sm mentions that although naming plays only a minor part in the grading of his students' work, he finds it important to always provide continuous feedback on naming conventions and good naming practices. This includes the active teaching approach of not evaluating a student's work if the names are "unreadable," or in other words, not to the standards that were taught in class. Only T4Sm specifies that naming practices are explicitly considered in the grade: *"During the projects, I assess how readable the code is. It's part of the readability, it depends on how they describe their variables. If they're all like x, a, z, and b, then I don't have a clue what's happening, so they'll get point reduction because it's not readable code. But, if they use the naming conventions that I've taught them and describe what's happening in the code then it's a lot more clear to read, so they'll get points for that."* He even implements specific naming assignments on the final test: *"there's a specific assignment in the test that's about what's happening in this program, and [I ask them to] rename the variables to make more sense [and] to be more descriptive" (T4Sm).*

Overarching patterns

We also investigated overarching patterns by grouping individual teaching approaches. The results are shown in **Table 3.5**. In short, we identified three different teaching profiles: (1) teachers that primarily use active approaches, (2) teachers that use a mix of approaches, and (3) teachers that hardly or do not at all incorporate naming practices in their courses. While teachers with an "active" profile show deliberate design choices for including naming practices, and those who do not teach naming practices either deliberately "opt out" of it, or were previously unaware that naming *could be* part of their course, most teachers show a "mixed" profile. This could indicate that teachers act intuitively based on their own experiences and beliefs regarding naming practices.

Our analysis further points towards a distinction between secondary and tertiary education: only secondary education teachers show an active approach to teaching naming practices, whereas university students are mostly expected to rely on their own abilities to learn and use appropriate naming practices. However, the small amount of teachers in our sample is not suited to draw any such conclusions definitively, and the distinction made here is purely based on the profiles emerging from the teaching approaches. As such, there is no clear indication (yet) that the teaching of naming practices requires different approaches across educational levels.

Table 3.5: Results of our cross-analysis into overarching patterns: three teaching profiles related to the teaching of the naming of variables are identified based on several common characteristics of our research questions.

Themes		**profile 1 - active**	**profile 2 - mixed**	**profile 3 - not taught**
	Dominant philosophy	naming is an essential skill	mixed, naming is learned by example	naming should not be taught
	Part of course	dedicated time allocated	no dedicated time, but woven into the course or by leading the way	no attention given, students rely on themselves
Active approach	*focus on conventions*	own guideline created	mixed	traditional conventions
	focus on readability	yes	mixed	no
	explicit naming assignments	yes	rarely	no
	attention to naming errors	yes	mixed	no
	naming tips and examples	on meaning and pro-actively given	mixed	on conventions and upon request
Passive approach	*required to follow guidelines*	yes	no	no
Own representation	*uses single letters or abbrev.*	no	mixed	mixed
	consistent with teaching	yes	mixed	mixed
Grading and feedback	*includes naming*	yes	no	no
	Teachers	T7Sm, T4Sm	T1Um, T2Um, T6Um, T8Um, T11Sf	T3Am, T5Um, T10Uf

3.4.3 RQ3: What do teachers observe in the classroom concerning variable naming?

Reported student observations

Teachers didn't observe many issues with naming practices among students. They tell us that naming practices are not a major source of concern and students 'get the hang of it' very quickly. For example, T2Um: *"We do insist on trying to give names that are as readable and as complete as possible, and I tend to believe [students] do that quite quickly. Apart from the first few sessions, where obviously they will use variables like 'x', 'y', 'z', and use names that don't say anything. We do insist on that during the practical sessions and in all the examples we give them. I think they very quickly catch on doing that."* Additionally, T1Um tells us once he points out a mistake, students often recognize their mistake immediately. Furthermore, students tend to use a mix of single letters, abbreviations, and single words in their first programs. However, throughout the course, and once students start to recognize and experience the importance of naming, they pick up the habit of using meaningful names (T2Um, T8Sm, T11Sf), and even start asking what convention they are expected to use at that point: *"After about three months of programming, and we start touching upon new items, students will start asking me themselves 'okay but what naming convention should we use for this thing'" (T7Sm)*. Also, T11Sf says, *"Currently, the students usually go back to 'a', 'b', 'x', 'y', and sometimes something more useful. And when they start PHP, of course not Python but PHP for their website, they start to notice 'o wait, the naming is kind of important'."* Furthermore, teachers observe that more experienced students choose more appropriate lengths for names (T11Sf), and, students that use better naming also present better programs in general (T7Sm). T5Um and T8Sm also note that their students seem to copy the examples that they are shown for their own naming practices. These observations are interesting in relation to our previous finding regarding the teaching strategy "lead the way": although teachers may not be directly aware of it, they seem to say that such a passive approach to teaching naming practices is valid, sufficient, and effective to teach naming practices.

Specific difficulties and reported causes

Mentioned difficulties were considered of minor importance by the teachers and we did not find any patterns among them. Firstly, teachers observe that most mistakes concerning naming appear because students are inconsistent, make typos, or lack creativity. T2Um notes that this might originate from an inconsistency between teachers, examples, and learning materials, which might further confuse students. Although the presence of typos could be just an oversight on the students' part, teachers mention this proneness to typos as a reason to not use too long names: mistakes are often and easily made in spelling, wrongly placed capitals, or using invalid names (T4Sm, T11Sf). T11Sf furthermore mentions that sometimes a student might lack creativity, possibly caused by a lazy attitude and a desire to make the assignments with the least amount of effort. Thinking of a good name might be considered "too much" effort, especially because students may not have been taught about "what is a good name". Teachers also observe confusion caused by names that are too similar, especially with longer names (T4Sm). Secondly, teachers note difficulties in connection to other identifiers such as functions and parameters (T2Um, T5Um, T10Uf, T11Sf). These difficulties include name conflicts or using the same names

for different objects, causing unintentional overwriting. Teachers attribute this issue to not being sufficiently introduced to the new concepts, meaning that with more practice and exposure, this mistake will disappear. Thirdly, teachers observe that students may (gradually) change names during debugging and writing (T1Um, T10Uf), leading to confusion when students have forgotten that names have changed, or when the name was only changed in one part of the code, but not yet in another. Finally, T6Um and T10Uf observe that there exists confusion among students about the "i" in a for-loop: students do not seem to understand that this is also a name.

3.5 Discussion

3.5.1 Impact on teachers and educators

The results from the interviews suggest possible impact for teachers and educators in three directions:

Teaching approaches for naming

While many teachers indicated that they mention naming conventions and guidelines, only teachers with an "active" approach indicated they use explicit pedagogic approaches that focus on the naming of variables in their classes. These teachers focused on an instructional approach with direct assignments or tips. At the same time, the interviews suggest that teachers realize that choosing a proper variable name is context-dependent and dependent on who will read the code. As a result, we see room for adopting a wide range of sociocultural teaching approaches that focus on group and peer activities. Especially considering teachers' philosophy in which naming is learned by example, some activities can include the use of live coding sessions, peer instruction-assessment-review, and pair programming, all with a focus on the naming of variables.

Developing CS Teachers' PCK on code quality

Teachers need to develop further their Professional Content Knowledge (PCK) on code quality in general and on the effect of naming on code comprehension. This is especially important for introductory courses that include learning objectives related to code quality. However, from the interviews, it seems that there is a matter of priority, as some teachers indicate that there are more important concepts to focus on than naming, especially when it comes to grading or feedback. To tackle this, the link between naming and code quality needs to be stressed. Having readable and expressive variable names is not a matter of code aesthetic, but rather an important aspect of code quality that is known to affect code comprehension [Lawrie et al., 2006, Schankin et al., 2018, Avidan and Feitelson, 2017]. The effect of bad naming will extend to the professional life of the student as a developer and will have an impact on their ability to contribute to projects and on the performance of daily programming tasks [Xia et al., 2018].

Using existing tools

Teachers are also capable of giving constructive feedback on naming. We believe that while teachers are obtaining more PCK on code quality, they could already implement such feedback. Prior work has found feedback on naming both desirable and valuable [Börstler

et al., 2017, Glassman et al., 2015]. Practical examples of existing teaching resources are rubrics and tools that are recently developed for such goals [Stegeman et al., 2014, Stegeman et al., 2016, Glassman et al., 2015]. These tools can be a good starting point to evaluate where students stand in their variable naming so that teachers can give constructive feedback to help improve the level of readability, expressiveness, and consistency of the naming of variables in their code assignments.

3.5.2 Reflection on teaching themes

When looking at the emerging themes on teaching variable naming, generated from the interviews, we can see that these themes follow the two mainstream theoretical pedagogic approaches in computer science education [Fincher and Robins, 2019]. On one side, the 'active' teaching theme follows an "instructivist" approach: the focus is on the structure and presentation of learning materials more than on the learners who are seen as recipients. Yet, this is not a pure picture: within the profiles of teachers who presented quotes fitting to the active teaching theme, we also observe aspects of "constructivist" approaches. In particular, some teachers refer to programming languages' guidelines on naming as a way to 'support that construction of knowledge' rather than to communicate knowledge. This, in effect, delegates learning goals to the students who will discover the topic of naming on their own and decide which names to use, without the teachers integrating their students' activities into the classroom. These and similar constructivist approaches of teaching variable naming are even more visible within the passive teaching theme, again with less focus on students' activities in the classroom. From the interviews, we observe that such teaching approaches have roots in the teachers' beliefs and perceptions that naming does not need explicit attention because it "comes naturally by example".

3.5.3 Limitations and future work

As our research is based on self-reported data, teachers may have given us socially accepted answers. We have tried to limit this by making the topic of the interviews more general about variables, formulating questions neutrally, and taking into account the order of the questions to avoid leading. Especially regarding teaching practices and student difficulties, our findings are self-reported: we did not observe classroom practices ourselves. However, to look at actual practices, we have conducted research into Massive Open Online Courses (**Chapter 4**) and Programming Textbooks (**Chapter 5**) and found similar results [van der Werf et al., 2023, van der Werf et al., 2024b].

Although there was no indication before this work that naming is or should be addressed differently across educational levels, our findings suggest this might be the case. However, as often with qualitative research, our sample set is too small to make representative conclusions, and being representative was not our current aim. Nevertheless, our study could be followed up with a large-scale (international) questionnaire to generalize and compare target audiences, class sizes, and class duration. Such research might also provide further insights into the different teaching profiles that we have found and could further dive into comparing naming practices among different programming languages.

The effect of the programming language

Some findings are specific to Python. For example, one difficulty that teachers described was that of students using the same name for variables and functions, causing unintentional re-assigning, which would not be a difficulty in statically typed programming languages. Furthermore, naming conventions and guidelines, which were often mentioned during the interviews, are to a large extent language-specific. Finally, characteristics of other programming languages not native to Python, such as pointers and static types, will not be reflected in our findings, even though they might have effects on teachers' perceptions and practices on variable naming.

Future work

Our future line of research is to analyze programming textbooks, to further understand how practices are represented in different forms of education. Additional research is planned covering in-class observations, which can be compared with what teachers say about the topic. We also aim to design and experiment with specific naming tasks to investigate how naming can be easily but effectively implemented in existing curricula.

3.6 Conclusion

This chapter aimed to investigate the current teaching practices and beliefs concerning variable naming. Primarily we want to encourage discussion on teaching naming practices in programming education. Hence we investigated teachers' beliefs and perceptions about variable naming (RQ1), their practice (RQ2), and their observations in the classroom concerning naming (RQ3).

Our results show a diversity of opinions; however, in line with most existing literature on 'good naming' for comprehension, our teachers all advocate for simple, straightforward, and intuitive names that clearly represent the content or show the purpose of the variable. Nevertheless, when it comes to the actual teaching practice, this promotion of meaningful names is not so directly demonstrated. Even though teachers tell their students to use meaningful names, they seem to rarely incorporate practices that encourage or force students to think critically about what a good name entails, or how names might be misleading. Moreover, teachers themselves do not always use meaningful names in their examples to students, even though they agree that students learn naming practices by example.

Interview protocol

Questions that explicitly cover the topic of NAMING of variables: 2b, 2c, 3, 4b, 6, 7, 10. Other questions might include naming if the interviewees brought up the topic themself.

Introduction (5 minutes)

- Overview, duration, recording, confidentiality, anonymity
- Introduction interviewer + interviewee

Practice (15-20 minutes)

(1) Can you shortly describe the setting of your course?

   a) Follow up on the level of education, online/offline, class size, language, duration

(2) Can you tell me something about how you explain variables in your course(s)?

   a) Can you give me an example?
   b) Follow up on topics related to variables (assignment, **naming**, role), dedicated time/attention, when introduced, etc.
   c) In your courses, what type of **names** are you promoting? Why? Motivate.
      i) Follow up on short & concise (abbreviations, letters) vs. full words
      ii) Follow up on examples of promoted names
      iii) Follow up on underscore vs. camel case
   d) If not taught: Why not? Can you provide a reason? (Is it a conscious choice?)

(3) Can you provide me an example of how you **name** variables yourself while you explain other concepts throughout your course(s)?

   a) Would you consider this example to be generic for the way you use variables in your teaching? (Why not? // Are there other ways that you use variables in your teaching yourself?)
   b) Follow-up on name length, letters, words, conciseness
   c) Follow-up on underscore vs. camel case

(4) (if time) Are variables evaluated in your course? Why? How?

   a) Can you give me an example?
   b) Follow up: formally/informally, which elements (inc. **naming**?), why (not)

Student difficulties (15-20 minutes)

(5) What are common errors that you see your students making when it comes to the concept of variables?
   a) Follow up on misconceptions identified, causes, how to overcome

(6) Can you give me some examples of how your students struggle when it comes to variable **names**? What difficulties do they experience?
   a) Follow up on why they might occur and how teachers solve them.

General perceptions (10 minutes)

(7) In your opinion, what should variable **names** consist of? What information should it contain?
    a) What do you consider "good naming"? What do you consider "bad naming"?
(8) As a programmer, and speaking in general, how important are variables to you while programming your own code and/or understanding someone else's code?
(9) As a teacher, how important do you consider variables for teaching programming skills to students?
(10) If you could make one recommendation to other teachers about teaching variables and their **naming**, what would it be? Motivate.
(11) Recommendations about what to stop doing?

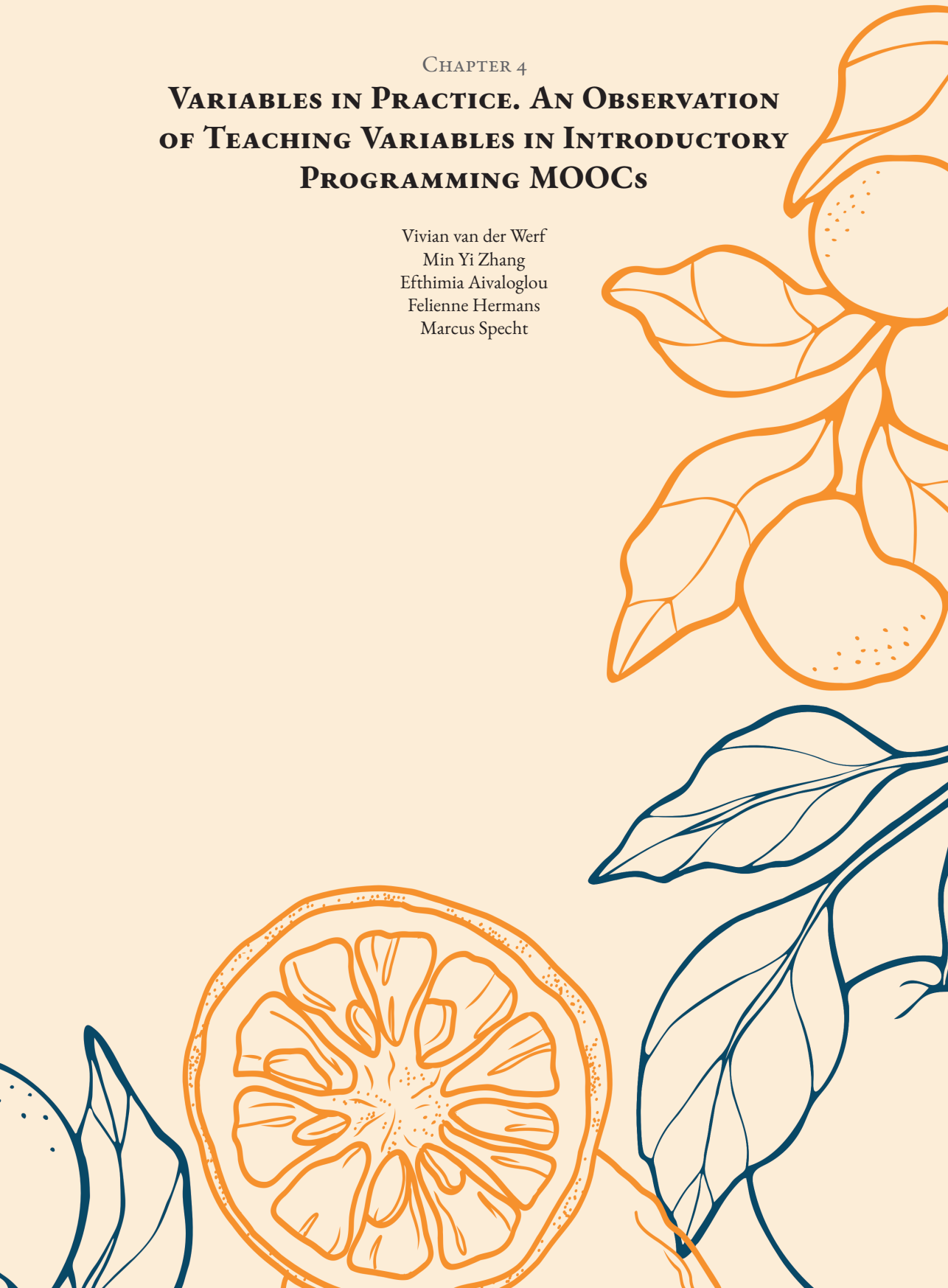
Closing

(12) Do you have final remarks or questions?

Chapter 4

Variables in Practice. An Observation of Teaching Variables in Introductory Programming MOOCs

Vivian van der Werf
Min Yi Zhang
Efthimia Aivaloglou
Felienne Hermans
Marcus Specht



Abstract

Motivation. *Many people interested in learning a programming language choose online courses to develop their skills. The concept of variables is one of the most foundational ones to learn, but can be hard to grasp for novices. Variables are researched, but to our knowledge, few empirical observations on how the concept is taught in practice exist.* ***Objective.*** *We investigate how the concept of variables, and the respective naming practices, are taught in introductory Massive Open Online Courses (MOOCs) teaching programming languages.* ***Methods.*** *We gathered qualitative data related to variables and their naming from 17 MOOCs. Collected data include connections to other programming concepts, formal definitions, used analogies, and presented names.* ***Results.*** *We found that variables are often taught in close connection to data types, expressions, and program execution and are often explained using the 'variable as a box' analogy. The latter finding represents a stronger focus on 'storing values', than on naming, memory, and flexibility. Furthermore, MOOCs are inconsistent when teaching naming practices.* ***Conclusions.*** *We recommend teachers and researchers to pay deliberate attention to the definitions and analogies used to explain the concept of variables as well as to naming practices, and in particular to variable name meaning.* [1]

Keywords

programming education
variables
naming practices
analogies
qualitative content analysis

[1]Published as: van der Werf, V., M.Y. Zhang, E. Aivaloglou, F. Hermans, and M. Specht (2023). *Variables in Practice. An Observation of Variables in Introductory Programming MOOCs.* In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, **ITiCSE 2023**, page 208–214, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3587102.3588857

4.1 Introduction

Variables are a hard concept to grasp for novice programmers [Hermans et al., 2018b, Gienow, 2017, Kohn, 2017]. At the same time, variables are important for reading and understanding code, which are both core skills for a proficient programmer [Pelchen and Lister, 2019, Corney et al., 2011, Lister et al., 2009, Lopez et al., 2008, Sajaniemi, 2002]. Variables are at the core of many programs, as they are able to store and retrieve data from memory ever since the introduction of "variable cards" in *The Analytical Engine*, which was designed in the 1830s and laid the foundation for modern-day programming. Many programming concepts expand on the concept of variables, for example, loops, functions, and control flow, and it is essential that variables are well understood by novice programmers. Furthermore, naming is an important aspect of variables, especially concerning the act of reading and understanding code. It is commonly accepted that meaningful identifier names help readers understand code more easily than when abbreviations or (random) letters are used [Avidan and Feitelson, 2017, Lawrie et al., 2006], although it has also been found that full names can be misleading if they do not correctly represent their contents or purpose [Arnaoudova et al., 2016, Caprile and Tonella, 2000].

Little empirical research has investigated *how variables are taught*, hence we are interested in conducting an observational study to gain insights into current teaching practices regarding variables. Since online platforms such as *edX* and *Coursera* grow increasingly popular [Koksal, 2020], we use Massive Open Online Courses (MOOCs) as a case study for our observation study. Our research questions are:

RQ1 How is the concept of variables taught in introductory programming MOOCs? We investigate (a) the connection to other programming concepts when variables are introduced, (b) how variables are defined, and (c) what analogies are used to explain variables.

RQ2 How are variable naming practices taught in introductory programming MOOCs? We examine naming practices that (a) are taught explicitly, and (b) are used by the instructors, and therefore taught implicitly.

During our analysis, we found that variables are often taught in close connection to data types, expressions, and program execution, and are often explained using the 'variable as a box' analogy. This represents a stronger focus on 'storing values' than on naming, computer memory, and the flexibility gained by using variables. Furthermore, we found inconsistencies in the taught naming practices – if they were taught at all. We recommend teachers and researchers to pay deliberate attention to naming practices, specifically to meaning, and to the definitions and analogies used to explain the concept.

4.2 Related work

Analogies are used to explain programming concepts [Fincher et al., 2020]. In education, an analogy, metaphor or notional machine is a 'tool' that supports learning by simplifying a concept through a representation that highlights the most important aspects of the concept, while obscuring less important aspects [Fincher et al., 2020]. For example, 'variables as parking spaces' simplifies the concept of variables, transferring our knowledge about parking spaces to our comprehension of a variable. Waguespack [Waguespack, 1989]

explains a variable of a particular data type as a 'container with the corresponding shape' (*shape* refers to the data type). In this explanation, a container can hold only a single value. This assumption is important since a common misconception is that variables can hold multiple values at the same time [Hermans et al., 2018b, Boulay, 1986, Chiodini et al., 2021]. Any analogy might only partly or incorrectly represent a concept and thus can leave novice students with an incorrect understanding or misconception. Nevertheless, Doukakis et al. [Doukakis et al., 2007] found that using an analogy still appears preferable over using none.

Thirty years ago, teaching variable naming was rarely included in programming textbooks [Keller, 1990]. There is no recent research on this topic, however, since then a considerate amount of work focused on the effect of naming on program comprehension, code quality and coding skills [Lawrie et al., 2006, Lawrie et al., 2007b, Avidan and Feitelson, 2017, Hofmeister et al., 2017, Cates et al., 2021, Blinman and Cockburn, 2005, Schankin et al., 2018, Sharif and Maletic, 2010, Binkley et al., 2009, Teasley, 1994, Beniamini et al., 2017]. Some results indicate that meaningful identifier names are most beneficial for code comprehension and debugging, and that "better names" are associated with better code quality. Few studies with a focus on naming aim at improving programming education. In the context of improving online curricula, Glassman et al. [Glassman et al., 2015] developed a tool and a quiz for their MOOC to assess naming on length and vagueness. They found that feedback on naming practices, as well as both good *and* bad examples, was highly valued by students. Other studies on variable naming in education found that novice programmers often fail to name variables correctly [Gobil et al., 2009] and that Scratch students are misled by variables named with a letter, probably because of prior knowledge from their mathematics education [Grover and Basu, 2017].

Observation studies on variable naming often target code quality and efficiency, and are based on names "found in the wild", meaning used by professional developers and/or taken from open source projects [Beniamini et al., 2017, Gresta et al., 2021, Newman et al., 2020]. Although Swidan et al. [Swidan et al., 2017] investigated naming practices in Scratch, a programming language for children, to the best of our knowledge no empirical observations investigate classroom practices on teaching variables or their naming.

4.3 Methods

To answer our research questions, we analyzed seventeen MOOCs on the platforms *Coursera* and *EdX* throughout March and April 2022. We searched for the MOOCs using the keyword *"programming"*, filtering for 'courses' and 'available now', excluding archived courses. Additionally, we applied the following selection criteria: the MOOC has to be (1) a beginner's course without programming prerequisites, that (2) focuses on (fundamental) programming skills and concepts. This information is obtained from the title and course descriptions. Furthermore, the course is (3) provided by a university, (4) taught in English, and (5) has at least one of its objectives to teach Python, Java, or C. Lastly, the course (6) should be freely available to anyone. Thus, courses that cover some programming but mainly focus on data science or web development are excluded, as well as courses that are created by companies.

Following the criteria, we looked at the first two pages of search results on both

platforms, as we argue that these courses are most likely to be chosen by those interested in learning the skill. Arguably, these are the most popular and relevant courses, with a significant number of students enrolled and high ratings (between 4.3 and 4.8 out of 5 stars). This led to the selection of seventeen MOOCs.

Of these seventeen MOOCs, seven are dedicated to Python programming (labeled P1-P7), six to Java (J1-J6), and four to C (C1-C4). One course (C1) teaches multiple languages, but as the C language was focused on most, we treat the MOOC as teaching C. Most of the courses are offered by US institutions, including prestigious universities such as Harvard (P3, C1), Princeton (J4), and the University of Pennsylvania (P6, J5). Only P5, J1, and J2 are respectively from Canada, Hong Kong and Spain. Ten MOOCs are taught by multiple instructors. Two teachers were (co-)teaching two MOOCs within our selection. In total there are thirty instructors.

To collect our data, we enrolled for the selected courses as would a regular student. The data that we used were: (1) the pre-recorded video lessons, (2) the lecture slides, (3) the practice materials and exercises, and (4) the additional explanations in between the video lessons. We used MS Excel to collect all relevant quotes, examples, and screenshots from the MOOCs. Data collection was carried out by the second author of this work. The first and second authors worked in close collaboration for the analysis, and uncertainties were discussed and resolved during regular sessions.

For RQ1a, we looked at the concepts explained right before, together with, and right after the introduction of the concept of variables. For RQ1b, we gathered the formal definitions given to variables and counted often recurring terms. We are specifically interested in how the definitions answer "what is a variable" and "what does a variable do". For RQ1c, we collected the analogies addressing the concept of variables, including visualizations that we found in the course material. For of RQ2a, we collected the explanations, tips, and explicit examples concerning variable naming practices. From these we established two categories: language rules, and human guidelines (conventions and variable name meaning). For RQ2b, we analyzed the variable names that were presented to students in videos, exercises and other learning materials.

4.4 Results

4.4.1 How is the concept of variables taught?

Connection to other programming concepts

Most often the analyzed MOOCs introduce variables at the beginning of the course; either right before, simultaneously with, or right after introducing data types and/or arithmetic expressions (see **Table 4.1**). In P3 and P5, variables and functions are explained together, and in C1 and C2, variables are introduced after functions, control flow, and loops. P4 introduces variables together with the concept of control flow and code order: *"Variables are possibly the most fundamental element of programming. There really isn't much you can do without [them]. (...) We use variables to represent the information in which we are interested, like stock prices or user names, and we will also use variables to control how our programs run, like counting repeated actions or checking if something has been found."*

Table 4.1: Concepts related to variables and when they are introduced. DT=data types; EXP=expressions.

Variables are introduced...	MOOCs
...right before DT and EXP	C4, J1
...together with EXP, usually before DT	P1, C3, J2, J3, J6
...together with DT, usually before EXP	P2, P4, C2, J2, J5
...right after EXP and DT, but before FUNCTIONS	P6, P7, J4
...together with FUNCTIONS	P3, P5
...after FUNCTIONS	C1, C2

Definitions

A formal definition of a variable is given by 12 MOOCs (70%), from which we can identify four recurring elements: variables (1) **store**, (2) **values or data** in (3) **memory**, and (4) can be referred to with a **name**. As shown in **Figure 4.1**, almost all definitions agree on what variables do (storing values or data), but there is no consensus on what a variable is (container, name, place in memory). Both memory and naming are less frequently included than "storing values". Only P1 and J3 explicitly cover a 'complete' definition, for example, *"A variable is a named place (4) in the memory (3) where a programmer can store (1) data (2) and later retrieve the data using the variable name (4)" (P1)*. An incomplete definition looks like *"A variable is just a container for some value (2) inside a computer or inside of your own program" (P3)*. Five MOOCs do not give a definition (P5, C4, J2, J5, J6).

An interesting finding is that, although not in the definition, C2 is the only MOOC to demonstrate that *flexibility* is also a major benefit, or even purpose, of using variables.

Analogies

Eight MOOCs (P3, P4, **P5**, C2, **C3**, **J1**, **J3**, **J6**) use analogies to explain the concept of variables. Of these, five (**bold**) also use matching visualizations. The identified analogies relate to the memory address and to the contents of a variable.

Most common is the analogy 'variable as a box', or variations of it, such as 'variables as a mailbox' (J1, **Figure 4.2a**). Boxes are often drawn with values inside and labels attached to the boxes (see **Figure 4.2**). The analogy also appears as only a visualization (P1) and without any visualizations but during 'live' coding: *"what we'd like to memorize is an integer value. Suppose I want to memorize this '17' right here. To do so, I need to first create a variable. So a memory box with room to store the 17 in. And then I need to place that 17 into this memory box" (C2)*.

Some instructors explicitly relate variables to the computer's memory. J1 introduces a figure (**Figure 4.3a**) to explain that the current value of a variable can be retrieved by referring to its name; in the illustration, an arrow is drawn from the variable name to this piece of memory. P5 draws a variable name with a box (**Figure 4.3b**) and emphasizes that the value does not go into that box but lives at a particular memory address. To clarify, he draws another square with the value '20', picking an arbitrary memory address marked with 'x3', explaining that the assignment statement takes 'x3', and puts it in the box associated with the variable name. P5: *"So 'base' contains 'x3' and (...) what that means*

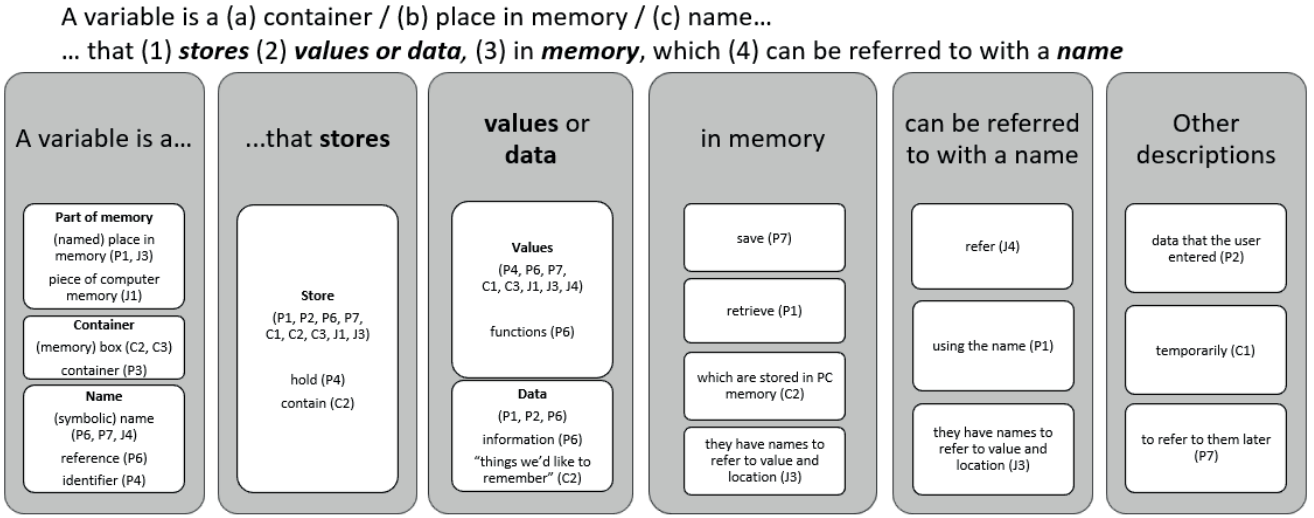
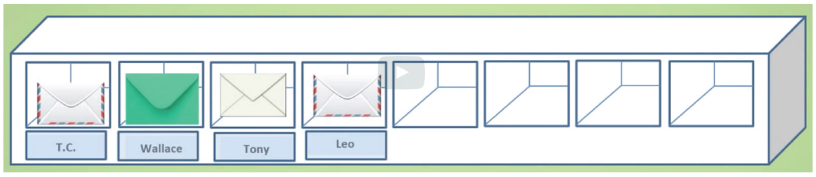
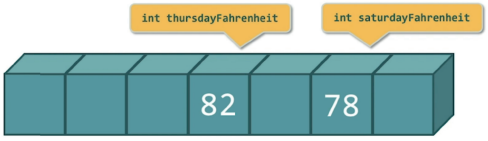


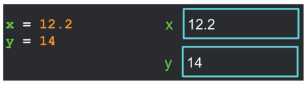
Figure 4.1: Analysis of variable definitions. The concepts are extracted from 12 definitions.



(a) J1, Variables as a mailbox. *"Each mailbox is labeled by its owner (or identifier) and different kinds of mails (or values) can be put into the mailbox"*

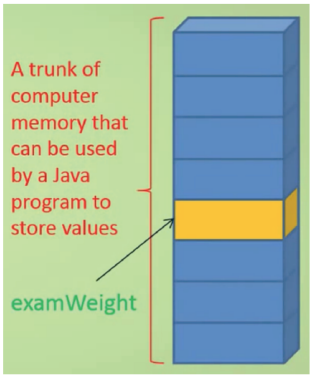


(b) J3, Variables as a labeled box



(c) P1, Variables as a box

Figure 4.2: Visualizations of the analogy "variables as a box"



(a) J1, memory

(b) P5, box analogy and memory

Figure 4.3: Visualizations connected to computer memory

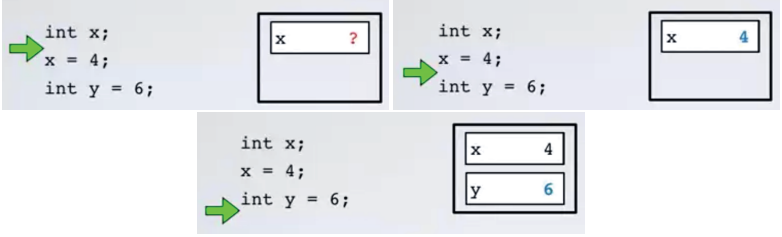


Figure 4.4: Three different stages of program execution (C3)

is that [it] points to the memory address 'x3' where the value '20' lives. (...) Python keeps track of [the variable's] value in that little box, and its value is a memory address." He continues *"[the memory address is] arbitrary, Python is in charge of that choice, and so I don't need to worry about exactly what the memory address is as long as I know that this relationship between variables and their values exists."*

A recurring visualization addresses the declaration and initialization of variables, by also covering the topics of program execution and tracing (J4, C3, J6, **Figure 4.4**). After highlighting the importance of drawing pictures of what happens during program execution, and in line with the box-analogy, the instructors of C3 and J6 create a box labeled with the variable name for a first statement, entering a question mark as the variable is still uninitialized. They then illustrate that executing a second statement will put a value in the previously created box. As a result, the question mark disappears.

Another analogy we encountered is that of a "question-and-answer" format, which connects well to variable naming practices. P4: *"If you're ever confused about what a variable means, treat it as a question."* As an example, the instructor shows that "num cats" can become the question "Number of cats?". *"The value, then, is the answer to this question."*

4.4.2 How are naming practices taught?

Which practices are taught?

We observe two primary aspects that are explicitly taught: (1) 'syntax rules' that *need* to be applied for the language to interpret correctly, and (2) 'human guidelines' that *can* be applied to aid a human interpreter. The first category, syntax rules, includes legal characters, reserved keywords, and case sensitivity. The latter category, human guidelines, breaks down into two subcategories: standardized *conventions* and the *variable name meaning*, both supporting code readability. An overview of which topic was represented per MOOC is provided in **Table 4.2**. MOOCs mostly focus on conventions and syntax whereas variable name meaning is covered in only half of the MOOCs, often superficially and with the interpretation of 'meaningful' varying per course and context. Three MOOCs do not cover naming practices at all and two more only cover syntax rules.

1. Syntax. Nine out of ten MOOCs covering syntax rules mention which characters variables may contain: i.e. in Python variable names cannot start with a number character or contain spaces. Four MOOCs mention that certain words are reserved keywords, such as 'if', 'for', and 'return', as explained by J3: *"To avoid confusing the compiler, you can't use reserved words as identifiers. Reserved words are words that are already given specific meanings in Java."* Finally, six MOOCs bring up case sensitivity, for example, P1: *"And it's case sensitive, but we don't want you to depend on that. So 'spam', 'Spam' with one upper case, and 'SPAM' all are different variable names, but you're not doing anybody any favor if you think that's being clever."*

2a. Conventions. Twelve MOOCs introduce naming conventions, such as using underscores or camel case words. For example, P5 states, *"every programming language has a set of conventions for how to choose a name, much like websites have a particular style and*

Table 4.2: Overview of taught variable naming practices, ordered by meaning, conventions then syntax. When meaning is discussed, conventions are always discussed too, but not the other way around. Syntax sometimes stands on its own.

MOOC	syntax	conventions	meaning
P1	x	x	x
P4	x	x	x
C2	x	x	x
C4	x	x	x
J1	x	x	x
J2	x	x	x
P2		x	x
P7		x	x
C1		x	x
P5	x	x	
J3	x	x	
J5		x	
P6	x		
C3	x		
P3			
J4			
J6			
Total (N=17)	10	12	9

layout. In Python most variable names use only lowercase letters with underscores to separate words, we call this pothole case." Three (P7, C1 and J2) also specify using capitalized words for constant variables only as a subtle visual reminder: *"Kind of like you're yelling, but it really just visually makes it stand out. So (...) like a nice rule of thumb that helps you realize, oh, that must be a constant. Capitalization alone does not make it constant, but [it] is just a visual reminder that this is somewhere, somehow a constant" (C1).*

2b. Variable name meaning. Nine MOOCs bring up the topic of meaningful or mnemonic names. They generally highlight that names are important for the readability of the code, but not so much for the language interpreter, as explained by P1: *"I emphasize that one of the key things about variable names is that you get to name them. We have a technique called mnemonic, and the idea is that when you choose a variable name, you should choose a variable name to be sensible. Python doesn't care whether you choose mnemonic variable names or not. The name that you choose for a variable does not communicate any additional information to Python (...) so mnemonic variables are only for humans."*

Even though teachers mention to use meaningful names, the interpretation of 'meaningful' varies between teachers and contexts. One example is choosing between letters and words. Some teachers find the use of letters as variable names not that meaningful, and rather replace 'x', 'y' and 'z' by *"more meaningful names such as 'radius', 'area', 'scores', if possible" (J1).* Also 'i' and 'j' could be more meaningful by using 'row' and 'column' in certain contexts, P7: *"Notice that if we chose 'row' and 'column', we'd immediately know by reading the names that they're indices. And more importantly, we'd know if we're looking at the vertical index or the horizontal index" (...) "I think that you'll find, if you pay a little*

bit of attention to your variable names, this won't be much of a burden, and it'll lead to significantly better code". P7 thus indicates that smart naming improves code quality and belongs to good programmer's practice. However, the instructor of C1 expresses that choosing a good name is all about context. In particular, in the context of arithmetic expressions ('x + y'), letters can be *"totally fine,"* as changing 'x' and 'y' to 'first number' and 'second number' *"isn't really adding anything semantically to help my comprehension" (C1)*.

One teacher (C4) spends extra time to highlight the importance of context when it comes to good naming practices. He first shows names that are usually considered 'bad', such as 'grx33', 'pp_25' and 'i_am_FourWords', pointing to that these names give no clue of what they might be, and the latter even mixes two conventions. However, he also says: *"we are not going to necessarily know if [the names are] good or bad, (...) they're going to have to have the right context. So 'data' may or may not be good, it may not be adequately descriptive. Maybe you need 'height data', or 'weight data', or something else."* He continues with several other examples, one of them being: *"(...) Again, if you were writing some movie with Superman and Mxyzptlk, [Mxyzptlk] might be an appropriate identifier."*

To summarize, about half of the MOOCs address variable name meaning, but what is considered meaningful, or 'good naming' is hard to define: it differs per instructor and context. Only the instructors of C1 and C4 acknowledge the importance of *context* when choosing a name and sought to clarify this by presenting several examples.

Which practices are used by teachers?

Most instructors apply the naming convention that is related to the language, so for Python and C that means lowercase letters and an underscore between words, and for Java it means using the camel case style. However, the instructors of MOOCs P4, C2 and C3 use the opposite style. Whereas the instructors of C2 and C3 do not give a clarification, P4's teacher tells the students: *"Each programming language has its own accepted style. In Python, you should use underscores. In Java and C#, you would use camel case. Other languages have their own conventions. 'But wait!' you say, 'You are using camel case in the videos!' That's right! I learned to program first in C++, and then in Java, and then in C#, three languages that use camel case instead of underscores. Old habits are hard to break!"*

Although nine MOOCs address the importance of meaningful variable names, only the instructor of P1 takes the students along to experience that importance. In the beginning of his course, he uses "silly" names such as 'eee', 'sval', 'xr' and 'nsv'. Later in the course, he uses examples such as 'count', 'largest_so_far', 'sum', and 'found'. He highlights that some names could be good names e.g. 'sval', 'fval' (string value, float value), but confusing to novices because they are unfamiliar with the terminology. However, in his explanations, he also stresses the unimportance of a name to the language interpreter, which reflects a stronger focus on 'names do not matter' than on 'what is a meaningful name': *"So you'll notice as I write, especially in these first two chapters, some of my codes use really dumb variable names and some of them use really clever ones. So I go back and forth to emphasize to you that the name of a variable, as long as it's consistent within a program, doesn't matter. And Python is perfectly happy" (P1)*.

The instructor of C2 chooses to 'lead the way' by using more meaningful names right

from the start all the way through to the end of the course. Examples of these names are 'age', 'balance', 'numberOfHazelnuts', and 'distanceTraveled'. In contrast, we observed that sometimes 'meaningless' names (for novices) are chosen to explain or show a specific concept, for example, C1 introduces the concept of variable scope with the help of 'foo'. Lastly, in almost all MOOCs, single letters, such as 'a', 'b', 'c', 'x', 'y', and 'z', are used to refer to variables holding numbers. Only the instructors of P4 and J1 do not use single letters and instead choose names such as 'aNumber', 'aInt", and 'aDouble'.

4.5 Discussion

We investigated teaching practices regarding the concept of variables and their naming, by systematically observing how variables are taught in introductory programming MOOCs.

We found that variables are usually taught together or in close connection with data types and arithmetic expressions, most often at the very beginning of the course. Furthermore, we recognized a pattern in how the concept of variables is defined. A definition is given in twelve MOOCs (70%) and centers around '**storing**' '**data**'. Some MOOCs also refer to a computer's **memory** and/or **naming**. Although there is a clear consensus on what variables do, we have seen no consensus on what variables are (i.e. part of memory, box, name). This might partly be due to the definitions of variables found in literature, such as 'containers that hold values' [Waguespack, 1989] and the origin of the concept from *The Analytical Engine*, where variables were used to store data. However, accessing, re-using, and modifying data is as important to the concept, but is little represented in our observation. Consistent with [Santos and Sousa, 2017], the 'variable as a box' analogy was often used while explaining variables. Since we already know this analogy might cause certain misconceptions [Hermans et al., 2018b, Chiodini et al., 2021], we suggest teachers should keep this in mind. We also found a questions-and-answer format as an analogy and some definitions that refer to variables as references. It would be interesting to further investigate how these latter analogies influence students learning, as well as the effect of a shift from 'storing data' towards other aspects of variables such as how and why we use them.

Naming practices are explicitly taught in most MOOCs, with syntax rules and conventions more often attended to than meaningful variable naming. Only half of the MOOCs allocate time to such naming practices, and when meaning is touched upon, a discussion on 'what is meaningful' is rarely implemented. Although plenty of provided examples concern syntactically acceptable names, few examples concern variable name meaning. This shows that not much seems to have changed since 1990 when Keller [Keller, 1990] established that choosing meaningful names for variables is rarely covered in programming textbooks. Moreover, our results could explain why many novice programmers fail to name variables correctly [Gobil et al., 2009].

When we look at implicit naming practices, in particular, how instructors use naming in the provided materials, we found that not all instructors used the naming convention style that is generally accepted for the respective language. However, Shariff and Maletic [Sharif and Maletic, 2010] indicate that underscore-styles versus camel case-styles do not influence a programmer's accuracy, which suggests that this inconsistency should not make much difference in student learning.

Furthermore, we found that instructors used different approaches regarding the meaning of a name. Only few MOOCs explicitly provided good and bad examples throughout the course or led the way from the start by always using names conveying the meaning of the content. In most MOOCs, letters were primarily used in demonstrations of code, and sometimes meaningless names were chosen to explain a certain concept. These findings reflect issues regarding naming practices in programming education. Firstly, they stress that meaningful naming practices are not very common in online education [Keller, 1990]. Secondly, the dichotomy in using letters as names, both when explicitly taught or solely used in the provided materials, is also reflected in the literature. For example, Lawrie et al. [Lawrie et al., 2007b, Lawrie et al., 2006] found that full word names are more effective for comprehension than letters, whereas Beniamini et al. [Beniamini et al., 2017] conclude that letters *can* be meaningful when they convey information that is commonly attributed to that letter. However, we observed that there sometimes exists no consistency in naming practices, which may leave students confused.

To conclude, it is becoming clear that appropriate variable names impact how quickly and how well a code is understood [Avidan and Feitelson, 2017, Beniamini et al., 2017, Binkley et al., 2009, Blinman and Cockburn, 2005, Cates et al., 2021, Hofmeister et al., 2017, Lawrie et al., 2007b, Schankin et al., 2018, Teasley, 1994]. We, therefore, might assume that, for this reason alone, they influence how learners learn a programming language, yet, still little is known about how these practices are taught and how exactly they influence learners. Based on our results, we feel a strong urge for both teachers and researchers to pay more attention to variable name meaning as part of naming practices.

4.5.1 Limitations

Since our study only covered the free-to-follow part of MOOCs, we did not include premium content features such as additional tests, assignments, or videos. It is possible that we have missed certain practices because of this, however, we wanted to examine what was available to everyone. Nevertheless, our answers on RQ2b, which naming practices do teachers use themselves, may have been influenced most, since it would have been valuable to see how instructors named their variables on the spot, a practice that was not always freely available. Furthermore, most of the MOOCs were created at US institutions, which might not be representative for online courses made and followed in other parts of the world. This is most likely an effect of our selection criterion that the MOOCs should be taught in English, or the fact that we used *edX* and *Coursera*. It would be interesting to compare our results with courses from other parts of the world, taught in local languages. Especially on the topic of variable name meaning, we would expect to see compelling variances.

4.6 Conclusion

We gained insight into how variables are taught in introductory programming education, in particular in MOOCs teaching Python, C, or Java. We found that the concept of variables is embedded through connections with other concepts such as data types, expressions, and program execution. There is a strong focus on storing data, whereas memory and naming are less well represented. Even flexibility as a benefit or purpose of variables is

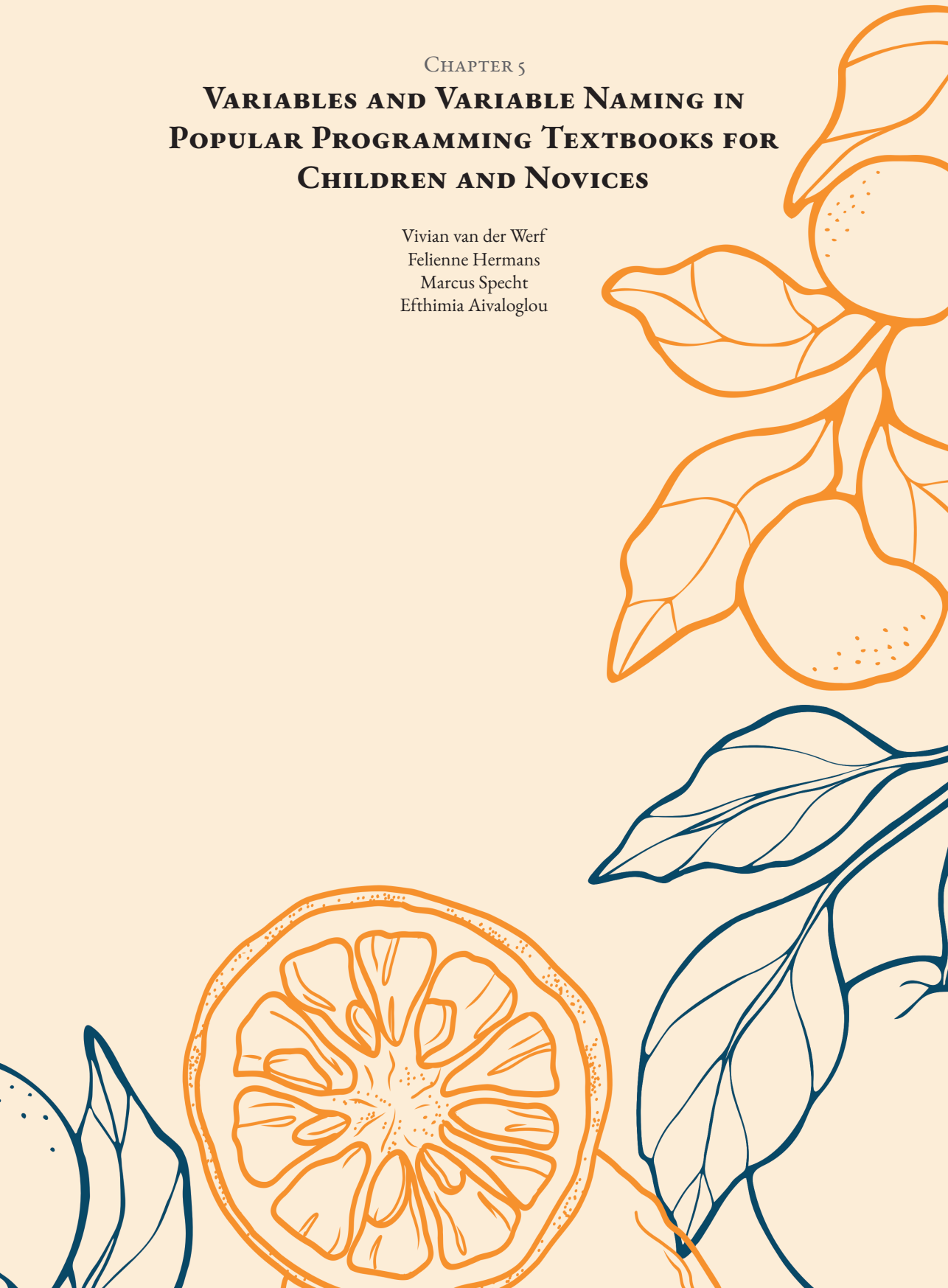
rarely mentioned. Furthermore, naming does not get consistent attention. Only a few MOOCs discuss the topic consistently with special attention to the meaning and context of names, whereas other MOOCs show inconsistency between taught and used practices, or show no discussion regarding meaningful naming at all.

Based on our results, we stress the importance for both teachers and researchers to pay more attention to naming practices, in particular to variable name meaning, and think about how these might influence the learning process of students. For future work we suggest extending our research by including observations from courses offered by tech companies and on YouTube, as many programmers might learn their skills there. Furthermore, we have conducted in-depth interviews with teachers of secondary-level and university-level education to complement the current research, with a special focus on naming practices (**Chapter 2**). Finally, it could be interesting to connect our results to known misconceptions, as suggested by [Hermans et al., 2018b].

Chapter 5

Variables and Variable Naming in Popular Programming Textbooks for Children and Novices

Vivian van der Werf
Felienne Hermans
Marcus Specht
Efthimia Aivaloglou



Abstract

In programming, the concept of variables is central to learning other concepts like loops, functions, and conditions, and the way variables are explained influences students' understanding. ***Chapter 4*** *observed Massive Open Online Courses (MOOCs) on introductory programming to investigate how the topic is addressed in teaching materials. This chapter aims to verify if these results generalize to other materials by analyzing 13 popular Scratch and Python programming books and investigating (1) which definitions and analogies are currently being used to explain the variables, (2) looking into the programming concepts that are introduced alongside variables, and (3) analyzing if and how variable naming practices are introduced. Our results support previous findings from MOOCs, suggesting that CS educators and developers of educational materials for introductory programming could pay more attention to how they explain variables and can be more deliberate and consistent when it concerns the teaching of naming practices. Additionally, we found specific analogies used to explain variables, and differences between programming languages in the order that variables are introduced. Our work can be used to update current educational materials and inform the development of new ones.* [1]

Keywords

programming education
variables
naming practices
analogies
programming concepts
qualitative content analysis
Python
Scratch

[1] Published as: van der Werf, V., F. Hermans, M. Specht, and E. Aivaloglou (2024). *Variables and Variable Naming in Popular Programming Textbooks for Children and Novices*. In Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, **SIGCSE Virtual 2024**, page 242–248, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3649165.3690112

5.1 Introduction

While variables are important for core programming skills such as reading and understanding code [Pelchen and Lister, 2019, Lister et al., 2009, Sajaniemi, 2002], they are also a hard concept to grasp for novice programmers [Hermans et al., 2018b, Kohn, 2017]. However, since several programming concepts expand on the concept of variables (i.e. control flow, functions), it is essential that variables are well understood. At the same time, variable naming practices are also relevant to the act of reading and understanding code: meaningful identifier names help readers understand code more easily than when abbreviations or (random) letters are used [Avidan and Feitelson, 2017, Lawrie et al., 2006]. Yet, other work also found that full names can be misleading if they do not correctly represent their contents or purpose [Arnaoudova et al., 2016, Caprile and Tonella, 2000].

Prior work [van der Werf et al., 2023] **(Chapter 4)** already investigated teaching practices regarding the concept of variables and variable naming by observing introductory programming MOOCs. To verify whether their findings generalize to other materials, the current chapter investigates the same topics in programming textbooks. Additionally, since previous work on textbooks [McMaster et al., 2016, McMaster et al., 2018] investigated *which* concepts are covered, but do not detail *how* these are covered, this chapter also aims to expand on the current state of knowledge on teaching practices. Following [van der Werf et al., 2023] **(Chapter 4)**, but in the context of introductory programming books, our research questions are:

RQ1 How are variables explained? (use of definitions and analogies)

RQ2 What other programming concepts are introduced either together with, right before or right after variables?

RQ3 How is naming addressed when variables are introduced?

5.2 Related work

5.2.1 Analogies for explaining variables

Analogies are often used to explain programming concepts [Fincher et al., 2020]. In education, an analogy, metaphor or notional machine is a 'tool' that supports learning by simplifying a concept through a representation that highlights the most important aspects of the concept, while obscuring less important aspects [Fincher et al., 2020]. For example, 'variables as parking spaces' transfers knowledge about parking spaces to the comprehension of a variable. Waguespack [Waguespack, 1989] explains a variable of a particular data type as a 'container with the corresponding shape' (*shape* refers to the data type). With metaphors like 'container' or the popular 'variables as a box', it is important however to stress that the container or box can hold only a single value. It has been found that, even though this analogy can support an initial understanding of the concept, it is also susceptible to the common misconception among novices that variables can hold multiple values at the same time [Hermans et al., 2018b, Boulay, 1986, Chiodini et al., 2021]. Any analogy might thus only partly or incorrectly represent a concept and can, therefore, leave novice students with an incorrect understanding. Nevertheless, Doukakis et al. [Doukakis et al., 2007] found that using an analogy appears preferable over using

none. More research is needed to understand which analogies are suited in which contexts, and if we should abandon the box metaphor entirely, perhaps replacing it with 'variables as labels' [Hermans et al., 2018b]. In introductory programming MOOCs, prior work [van der Werf et al., 2023] **(Chapter 4)** found common use of the metaphor variables as a (mail)box in explanations and visualizations and also a 'question-and-answer' format to think about variable names and contents. Another promising way of introducing and teaching variables is with the help of Sajaniemi's theory of "roles of variables" [Sajaniemi, 2002, Sajaniemi and Kuittinen, 2005], which categorizes variables based on their dynamic nature, i.e., *fixed value, stepper, gatherer*, or *most-wanted value*.

5.2.2 Variable naming

That (variable) naming is important for comprehension and code quality is indisputable from the existing literature focusing on the effect of naming on program comprehension, code quality, and coding skills. Most importantly, programmers rely on names for their understanding of code [Avidan and Feitelson, 2017, Hofmeister et al., 2017, Teasley, 1994, Takang et al., 1996, Lawrie et al., 2007b, Lawrie et al., 2006], and names often serve as beacons during code comprehension [Gellenbeck and Cook, 1991]. Moreover, bugs are easier to find when words are used [Hofmeister et al., 2017]. Additionally, names that are not descriptive enough, for example, single letters or abbreviations from which meaning is not directly clear, interfere with code comprehension [Lawrie et al., 2007b, Lawrie et al., 2006, Hofmeister et al., 2017, Beniamini et al., 2017]. The same holds true for names that are too long, making them difficult to remember [Binkley et al., 2009]. Additionally, names can be unintentionally misleading and should therefore be chosen cautiously [Avidan and Feitelson, 2017, Arnaoudova et al., 2016, Feitelson, 2023, Feitelson et al., 2022]. Especially general, non-specific names, such as 'length' [Feitelson, 2023] or 'result' [Schankin et al., 2018], appear problematic. Finally, novices can wrongly believe that computers interpret or assign values based on the semantic meaning of variables' names, and thus incorrectly apply semantic assumptions to syntax [Kaczmarczyk et al., 2010].

Consequently, it is relevant to think about how we teach variable naming in introductory programming courses. Thirty years ago, Keller [Keller, 1990] indicated that variable naming was rarely included in programming textbooks. Since then, little research observed teaching practices on this topic. Two recent studies [van der Werf et al., 2023, van der Werf et al., 2024c] **(Chapter 3, 4)** found that teachers do address naming practices in their learning materials, but inconsistently: variable naming practices are not always taught explicitly, taught practices are sometimes conflicting, and given example code does not always match the provided rules and recommendations. Moreover, research investigating code quality perceptions among students and teachers [Börstler et al., 2017] confirmed students' desire for 'more and more specific feedback about what was good and bad in their code'. Other studies on variable naming in education found that novice programmers often fail to name variables correctly [Gobil et al., 2009] and that Scratch students are misled by variables named with a letter, probably because of prior knowledge from their mathematics education [Grover and Basu, 2017].

5.3 Methods

To answer our research questions, we analyzed thirteen textbooks that aim to teach Scratch *or* Python to children and novices. To systematically select programming books we used two Amazon best sellers lists (top 100 popular products based on sales), both visited on April 18, 2023. For Scratch books, we selected the five books ranked highest within the *Amazon Best Sellers: Best Children's Programming Books*. Also for Python books, we selected the five books ranked highest within the same list. However, since teens and young adults might prefer using adult textbooks, we also added the three books ranked highest within the *Amazon Best Sellers: Best Python Programming*. For all lists the following selection criteria were applied: 1) being a physical book, 2) written in English, and 3) focused *solely* on learning Scratch or *solely* on learning Python. The selected books and their details are presented in **Table 5.1**.

Table 5.1: Overview of selected programming books

ID	Target	Bestseller	Title	Year
S1	Children	#6	Coding Games in Scratch [Woodcock, 2015]	2019
S2	Children	#13	Coding Projects in Scratch [Woodcock, 2016]	2019
S3	Children	#14	Code Your Own Games! [Wainewright, 2020]	2020
S4	Children	#22	Coding for Kids Scratch [Highland, 2019]	2019
S5	Children	#60	Learn to Program with Scratch [Marji, 2014]	2014
P1	Children	#4	Coding for Kids python [Tacke, 2019]	2019
P2	Children	#7	Python Coding for Kids Ages 10+ [Makda and Mamazai, 2022]	2022
P3	Children	#8	Coding Games in Python [Vorderman et al., 2018]	2018
P4	Children	#12	Python for Kids [Briggs, 2023]	2023
P5	Children	#19	Coding Projects in Python [Vorderman et al., 2017]	2017
P6	Adults	#2	Python Crash Course [Matthes, 2023]	2023
P7	Adults	#4	Python Programming for Beginners [Robbins, 2023]	2023
P8	Adults	#6	Automate the boring stuff with Python [Sweigart, 2020]	2019

To systematically collect our data and ensure good operational definitions, the first author created a a codebook in a Microsoft Form, which was tested on three random books (one from each category) by the first author and an independent data collector. Issues were resolved and a new version of the form was designed by the first author. This version was then independently used by both parties to gather all information relevant to the research questions. The data collector was recruited from a pool of research assistants and hired to reduce bias in the collection of data. As such, after transferring the data to MS Excel, the first author compared the two sets. Any information found by only one collector was reassessed for inclusion.

Each research question covered different chapters and was analyzed separately, as specified below:

RQ1: Explanation of variables We collected all definitions (quotes) and analogies (quotes and pictures) from the section in the book that introduces the concept of variables. We then also checked all other sections, and, when applicable, glossaries, for any definitions of variables. For example, sometimes a summary with a definition was also given at the end of a chapter, which was included.

The collected definitions were analyzed on the object (what are variables: nouns, i.e., a 'box', a 'memory location'), the purpose (what do variables do: verbs + addition, i.e., 'store information'), any additional information that was provided (i.e., 'data can change'), and if any, used analogies. For each definition, the relevant information was recorded. Additionally, when images were provided to accompany the definition, they were described and it was recorded which analogy it represents. The independent data collector and first author had no disagreements.

RQ2: Other programming concepts To investigate how variables are connected with other programming concepts, we examined the concepts discussed right before and right after the concept of variables. To this aim, we investigated three chapters: the chapter in which variables are introduced, the chapter before, and the chapter after. This means that if a topic is not represented in our results, it was either not covered in the book, or it was introduced in other chapters and therefore not considered in the analysis.

To collect the different concepts, we first made a list of expected programming concepts (based on [van der Werf et al., 2023] **(Chapter 4)**) to check for in the chapters and added to this list when we encountered a different concept. We then systematically analyzed the different chapters for the presence of these concepts. For the analysis, we categorized the concepts into the following topics: data types, operators, control flow, print-input statements, and others.

RQ3: Naming rules and guidelines To search for naming rules and guidelines, we looked at the chapter where variables were introduced. Any rules discussed here were collected, following the categories found in [van der Werf et al., 2023] **(Chapter 4)**: (1) *syntax rules*, including case sensitivity, accepted symbols, reserved keywords, and restriction of spaces; (2) references to specific naming *conventions*, such as camel case or underscore styles, and (3) any guidelines on *variable name meaning*.

For the first two categories we collected which rules and conventions were mentioned much like a closed coding process. For the third category, we used an iterative and open coding process which meant we analyzed the books several times. Based on an initial glance at the chapters, we first collected whether one of the following topics was addressed: 'use descriptive/meaningful names', 'avoid single-letter names', 'avoid misleading names', and 'you should be able to understand your name'. During this phase, we also gathered other quotes or statements on naming we encountered, if any, such as 'use a simple naming method,' 'too long names are hard to read,' 'consistency in naming is important,' and 'you can use any name you want.' Then, after going through each book, we went through all the books again to see if any newly encountered statements were missed in earlier books. To continue the analysis, we grouped all naming statements and quotes into four subtopics: those (a) suggesting to use meaningful/descriptive names, (b) addressing reasons for using such names, (c) addressing the length of the name, and (d) highlighting that names can be whatever you like. We furthermore noticed several other interesting quotes that were collected under 'other'. After the grouping, to ensure a complete overview, all books were checked a final time for any additional input on any of these four topics.

Besides this, we collected and investigated explicit examples and naming exercises, when provided. We then checked other parts of the books to see if naming was (also)

addressed elsewhere, for example, some books include a section or chapter on "how to improve your code". If naming was addressed elsewhere in the book, we recorded the context.

5.4 Results

5.4.1 How are variables explained?

Most Scratch books explain **variables as a box**, as opposed to only 3/8 Python books (see **Table 5.2**). This analogy is often accompanied by a picture that affirms it. A typical definition looks like 'a variable works like a box that you can store information in, such as a number that can change' (S1). Some books explain **variables as a place** or (memory) location, for example, a variable 'describes a place to store information, such as numbers, text, lists, and so on' (P4), or, 'a variable is a named area of computer memory' (S5). Few books (also) explicitly address **variables as a label**, for example, a variable is 'a fancy name or a tag' (P1) or 'essentially a label for something' (P4). Others include it more implicitly, mentioning that the variable is a 'labeled box' or needs a name 'to label the information.' To address the common misconception that often happens with the variable as a box analogy, P6 writes: 'Variables are often described as boxes you can store values in. This idea can be helpful the first few times you use a variable, but it isn't an accurate way to describe how variables are represented internally in Python. It's much better to think of variables as labels that you can assign to values. You can also say that a variable references a certain value.'

Table 5.2: Explaining variables with analogies and purpose.

Variables...	**Scratch**	**Python**
...as a box **with image*	S1, S2*, S3*, S5*	P3*, P5*, P8*
...as a place	S4, S5	P2, P4, P8
...as a label *(implicit)*	*(S2, S5)*	P1, P4, P6 *(P3, P5, P8)*
To store information	S1–S5	P3–P8
To track information	S2	P1, P3, P5
To access information	S5	P3, P4, P5
To interact w/ information		P7
To support code writing		P4
To use later		P3, P4, P5
Their value can change	S1, S2, S4, S5	

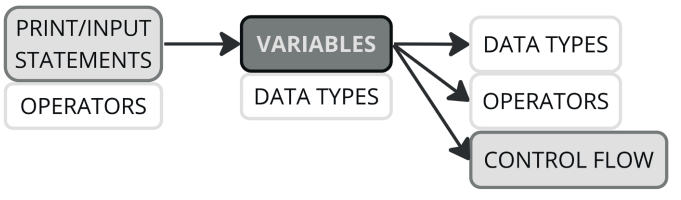
Most explanations address the purpose of variables. Books most often write that variables 'store information'. Other purposes mentioned are to 'keep track of information', 'to access information', 'to interact with information', 'to support code writing', 'and to use later'. In addition, only Scratch books mention explicitly that a variable's information (value) can change, for example, 'notice that the value of the score changes throughout the program. This is why we call it a variable – its value changes' (S5).

Scratch books primarily explain variables as a box; Python books use more diverse explanations. The emphasis is on 'storing information', while other purposes of variables get less attention. Only Scratch books explicitly mention that a variable's value can change.

5.4.2 What programming concepts are introduced alongside the concepts of variables?

Scratch and Python books apply different trajectories when it comes to which programming concepts are introduced alongside variables (see **Figure 5.1**). Below we discuss detailed results per concept.

(a) Scratch



(b) Python

Figure 5.1: Trajectory of programming concepts as introduced by (almost) all [in gray] or about half [in white] of the books.

Simple data types (string, integers, float, boolean) are discussed by all Python books, either in the same chapter as variables (P1, P3, P5, P6, P8) or in the next (P2, P4, P7). Only one Scratch book (S5) introduces them (right before variables). More **complex data types** or structures (arrays, lists, dictionaries, tuples) are covered in 4/8 Python books, either in the same chapter (P3, P5), and/or the next (P3, P6, P7). They are not addressed in Scratch books.

All Python books and three Scratch books address **mathematical operators** (+, -, /, *) in the predefined chapters. While most books prefer to introduce them *after* variables (S2, S3, S5, P1, P2, P6), Python books also introduce them right *before* (P4, P7, P8) or *together with* (P3, P5) variables. We see a different pattern for **comparison operators** (==, !=. <, >) and **logical operators** (and, or, not): they are introduced by almost all Scratch books and only half of the Python books, most frequently *after* variables are introduced, either in the same chapter (S2, P3, P5) or the next (S3, S5, P1, P8). Only S4 introduces comparison operators right before variables.

While all Scratch books introduce control flow concepts like **if-else statements** (5/5) and **loops** (4/5) in the predefined chapters, only three Python books do so. Moreover,

Scratch books (except for S5), introduce them *before* variables, while all three Python books (P3, P5, P8) introduce them *after*. We noticed that in Python books, control flow is often introduced later in the books.

Several other concepts are (sometimes) introduced in the predefined chapters. Here we mention only those that are covered in two or more books. For Scratch, these concepts are user input, random numbers, first program, error messages/bugs, and procedures. For Python books, these are print-statements/first program, using comments, error messages/bugs; user input, and functions/classes.

*Scratch books introduce mathematical operators after variables; Python books **also** introduce them right before or together with variables. Control flow concepts are introduced before variables in Scratch books, and after variables in Python books, if at all in the chapters surrounding variables.*

5.4.3 How is naming addressed?

Variable naming is addressed in almost all books, except for S3 (see **Table 5.3**). Nine books provide a dedicated section on naming, whereas three books only briefly mention naming. Three books provide dedicated naming exercises. Especially Python books also address naming in chapters, in the context of functions, scope, name errors, conventions, or readability. When naming is reintroduced, books mostly repeat what is mentioned in the chapter on variables, or explicitly refer back to it.

Table 5.3: Overview of how naming is addressed.

	Scratch	Python
Is naming addressed in the chapter that introduces variables? *(N)*	**4**	**8**
yes, naming is briefly mentioned	S2, S4	P4
yes, naming has a dedicated section	S1, S5	P1–P3, P5–P8
Are naming exercises provided? (yes)	–	P1, P2, P8
Are there explicit examples of "good" or "bad" naming? *(N)*	**4**	**8**
yes, but only on syntax rules ('valid' or 'invalid' names)	S4, S5	P7, P8
yes, the good examples also address the descriptiveness of names	S1, S2	P1–P6
yes, the bad examples also address the descriptiveness of names	–	P1–P6
Is naming addressed in other parts of the book? *(N)*	**1**	**6**
yes, when functions are introduced	–	P1, P3, P5, P7
yes, in a section on readability / conventions	S1	P2, P8
yes, in another section	S1	P2, P3, P5, P8

Syntax rules

Syntax rules, like reserved keywords and case-sensitivity (see **Table 5.4**), are addressed by all Python books (8/8) and just 2/5 Scratch books (S1, S5). Both Scratch books mention that spaces are technically allowed, but it is better to avoid them because other languages do not allow it, for example, *DogSpeed* instead of *dog speed* (S1) and *SideLength* instead of *side length* (S5). Besides mentioning the rules, several specific example names are also given in 5/8 Python books, for example, *Good1* (P2) is accepted, whereas *2Good* (P2), *100_days_of_code* (P1), and *TOTAL_$UM* (P8) are names disrespecting the rules.

Table 5.4: Overview of syntax rules addressed in the books.

Syntax rules	Scratch	Python
Names are case-sensitive	S5	P3, P5, P8
Use Unicode-letters, no symbols		P2–P8
Do not start with a number	S5	all
Do not use spaces	S1, S5	all
Do not use reserved keywords		P2, P4–P7

Naming conventions

The same two Scratch books (S1, S5) mention naming conventions, such as using underscores to separate words. All Python books refer to such conventions or 'community guidelines' (P7), although the specific conventions mentioned vary and can deviate from the common underscores, for example, camel case (P1, P2, P8), Pascal case (P1), Hungarian notation (P2), PEP (P8), and *Zen of Python* (P6, P7, P8). P2 and P8 also use camel casing in their example code. Furthermore, P2, P3, and P6 mention that constants are fully capitalized: 'constants will be named in all caps to spot them easily' (P2), for example, *PI* or *SPEED_OF_LIGHT* (P2).

Variable name meaning

Three out of five Scratch books note that it is preferable to use meaningful (S1, S5), sensible (S2), or descriptive (S5) names that 'tell you what the variable is for' (S1) and 'to make the code readable' (S1). Given examples are *speed, score, dragon* (S1), *High Score, Player Name* (S2), *firstName*, and *interestRate* (S5). On the other hand, 7/8 Python books instruct students to use descriptive (P1, P3, P5, P6, P8), meaningful (P2, P3, P4), or useful (P4) names. For example: 'When naming a variable you want to be as descriptive as possible but also follow the rules of Python (P1),' and, 'the variable name should be meaningful e.g. if a variable stores the name of my friend, then the variable name should be *friendName* not just *name* which can be confusing or misleading' (P2). Avoiding confusion is not the only reason given for using descriptive names. The idea of variables (and names) *storing* something *inside* them (see also Section 5.4.1) is again highlighted. For example, P5 writes to 'think of a name that will remind you what's inside the variable', others note that a good name 'describes the data it contains' (P3, P8). The most common argument, however, is to improve *readability*, explicitly mentioned by S1, P1, P7, and P8. Interestingly, the latter addresses its own examples as too generic: 'most of this book's examples use generic variable names like *spam*, *eggs*, and *bacon*, but in your programs, a descriptive name will help make your code more readable' (P8). Finally, two books note that good names will help you to *understand* (P1, P5) the code.

In 5/13 books, the length of variable names is also related to a name's meaning (S5, P3, P4, P5, P6). For example, S5 writes to avoid using single-letter names such as *w* or *z*, 'unless their meaning is very clear'. The book also continues with that 'names that are too long can make your script harder to read.' P6 stresses that names 'should be short but descriptive [therefore] *name* is better than *n*, *student_name* is better than *s_n*, and *name_length* is better than *length_of_persons_name*.' P3 and P5 are less explicit but give examples such as using *attempts* rather than *a* (P2) and *lives_remaining* rather than *lr* (P5). On the other

hand, P4 writes: 'Sometimes, if you're doing something quick, a short variable name is best. The name you choose should depend on how meaningful you need the variable name to be,' however, no further explanation is provided besides '*Fred* probably isn't a very useful name.'

Finally, some books (S4, S5, P4, P8) make an explicit mention that variables can be named anything. Whereas most do so while highlighting that descriptive and meaningful names are highly recommended, S4 only writes: 'you can name a variable anything you want–get creative (...) *points*, *goals*, or yes, even *hippo farts*.'

Regarding naming practices, most books focus on syntax rules that, when not adhered to, break the program. Python books give more attention to naming guidelines and variable name meaning, yet, like Scratch books, also present conflicting information, take a 'free-for-all' approach, or remain vague on what is a 'meaningful' name.

5.4.4 Patterns between children and adult books

The analysis highlighted two differences between the Python books for children and adults. First, in the books for children, the topic of functions was sometimes introduced within our predefined chapters, however in adult books this topic had a chapter elsewhere. Second, regarding naming, we found that all children's books and P6 provided explicit examples of "good" or "bad" names that cover what is and is not a descriptive name. The other two adult books focused on examples regarding syntax rules.

5.5 Discussion

We investigated how the concept of variables, and the respective naming practices, are taught in thirteen popular introductory Scratch and Python programming textbooks. Our collected data was qualitative in nature and included definitions and analogies used to explain variables, other programming concepts introduced with or near variables, and any naming practices that are addressed. Our most important findings are:

5.5.1 Variables are commonly explained as a box

From the literature, we know that analogies come with a risk of carrying over misinformation from one topic to the other [Boulay, 1986, Chiodini et al., 2021, Hermans et al., 2018b]. Consistent with prior work [van der Werf et al., 2023] **(Chapter 4)**, we found a tendency to explain variables as a box, which is prone to cause misconceptions when learning new programming concepts. Nevertheless, one book explicitly addresses this issue, while others opt for alternative explanations, such as variables as a label or place. This might indicate that the community is looking for new analogies, however, the consequences of these are yet to be investigated [Hermans et al., 2018b]. We also found most explanations to focus on 'storing information', which is again consistent with prior work [van der Werf et al., 2023] **(Chapter 4)**. Few other purposes of variables were mentioned, including tracking information, accessing information, and the ability to flexibly reuse data elsewhere in the code.

Hence we see room for using a wider variety of definitions and analogies and extending

the explanation to include different purposes. Domain isomorphic analogies [Bettin and Ott, 2023, Bettin et al., 2023], which are flexible in use across domains while preserving the analogical mapping, and roles of variables [Sajaniemi, 2002, Sajaniemi and Kuittinen, 2005] might be promising directions, keeping in mind students' background and cognitive load.

5.5.2 The concepts introduced near variables vary

Like prior work [van der Werf et al., 2023] **(Chapter 4)**, we found that variables are often taught in close connection to *data types*, *operators* (arithmetic expressions), and *control flow*. Additionally, we found that Scratch and Python textbooks introduce different programming concepts alongside variables. The order in which these concepts are introduced also differs between Scratch and Python books and among Python books. This raises questions such as when is the best moment to introduce variables, is there a "one-size-fits-all" trajectory crossing audience and programming language, or should such learning trajectories naturally depend on the programming language (and audience). Rich et al. [Rich et al., 2017, Rich et al., 2022] advocate for a language-independent learning trajectory focused on variables. However, our results hint towards current learning trajectories being influenced by language. This then also raises the question of how different trajectories influence transfer from Scratch to Python (or another programming language). Moreover, the variations we found within Python programming books suggest that a single "natural" trajectory, as we found for Scratch books, might not exist for Python. Alternatively, there might be unclarity or disagreement among developers on what order is most desirable, for example in terms of prior knowledge, avoiding or tackling misconceptions carried over from other disciplines or languages, or varying teaching purposes or learning philosophies. If the order of concepts was chosen carefully by the books' authors, there is an opening to investigate underlying motivations.

5.5.3 Naming is addressed inconsistently

In line with related work [van der Werf et al., 2023, van der Werf et al., 2024c] **(Chapter 3, 4)**, we also see that when naming practices are introduced, most books focus on syntax rules that, when not adhered to, break the program. Sometimes community guidelines and naming conventions are mentioned, but these are not consistent between *and* within books, therefore some books even provide conflicting information. Although the effects of style and casing on a programmer's accuracy might be limited [Sharif and Maletic, 2010], inconsistent approaches could confuse a learner, or unintentionally undermine the development of a critical attitude towards naming.

A careless attitude can be further encouraged in a learner by unclear definitions or examples of what is a 'meaningful' name. We have seen most books telling their reader to use meaningful or descriptive names, but some without indicating why naming is important. Moreover, some of those do not give explicit examples of what is considered meaningful, mention that variables can be named anything, or use generic variable names themselves. The limited attention to what is meaningful could be explained by that developers of educational materials chose a 'constructivist' pedagogical approach, in which students themselves discover by example what is good naming [van der Werf et al., 2024c] **(Chapter 3)**. In fact, two Python books (P2, P8) hint at using such an approach, writing that with

experience 'you will naturally know how to name [variables]'. However, for students to learn by example, we would expect the given guidelines and examples to be more consistent with each other. Perhaps we would even expect more emphasis on why naming is important rather than on certain rules and guidelines. Any inconsistencies, together with a limited explanation of why naming is important, could insinuate that one does not need to pay attention at all to naming practices.

Therefore, we suggest that our results demonstrate a potential misalignment between developers of educational materials and what research already knows is important for comprehension. Because our results are in line with prior work [van der Werf et al., 2023, van der Werf et al., 2024c] **(Chapter 3, 4)**, we suggest that if we want students to adopt good naming practices and develop a critical attitude, developers of educational materials *and* practitioners pay attention to how they address naming practices and be consistent in their approach. Moreover, considering that naming is context-dependent, there is room to focus on what makes a name (in)appropriate and why.

5.5.4 Limitations

Since our research analyzed only a limited number of books, our results might not be representative. However, by selecting the most popular books from Amazon, we aimed to include those books that people are most likely to buy and be exposed to, now and in the (near) future. However, even though Amazon is a popular platform, we cannot say if these bestsellers represent the books children and adults are truly exposed to. Using other (local) platforms or renewing the search at a different time might result in a different selection of books and hence influence our results. Nevertheless, the results we found correspond with results from prior studies, which suggests that our selection of books is reasonably representative. Even so, since most of the books included in this study were published relatively recently, older books, which could be designed differently, may likely still be in use. Finally, Scratch and Python are the languages most used by children. Had we focused on adults, other programming languages should be taken into account. We expect some differences due to the nature of the language, just like we found between Scratch and Python.

5.6 Concluding remarks

Our observations strengthen existing insights into how variables are presented in programming MOOCs, and extend them to programming textbooks for children and novices. More attention in research is needed to, for example, when to introduce the topic within the curriculum (in which language). Our insights also call for a (more) careful approach regarding variables and their naming, to be taken by educators and developers of learning materials in the fields of Computer Science and Software Engineering. Most importantly, we encourage the community to use (1) a wider range of definitions and analogies while teaching the concept of variables and (2) a more consistent teaching approach regarding variable naming that goes beyond syntax rules, personal preferences, and naming conventions. This includes a discussion on the importance of the topic and what makes a name (in)appropriate and why.

Chapter 6

Promoting Deliberate Naming Practices in Programming Education: A Set of Interactive Educational Activities

Vivian van der Werf
Felienne Hermans
Marcus Specht
Efthimia Aivaloglou

Abstract

Despite extensive studies from the software engineering community on how naming practices influence programming behavior, the topic receives little attention in education. Prior work indicated little agreement on good naming because it depends on many factors. Students are told that "naming is important" and "should be meaningful," yet its practical implementation is rarely discussed and feedback is lacking. The current work presents a dialogic teaching approach focused on teaching a critical reflection on naming practices through five activity types: (A) perceptions and experiences, (B) create names, (C) evaluate through ranking, (D) compare codes, and (E) locate a mistake. We developed, ran, and analyzed a one-hour workshop, that we present here and share our experiences, leading to recommendations for teachers. Our contribution is twofold: (1) we provide a set of (adaptable) activities and exercises for supporting deliberate naming practices, thereby assisting teachers interested in adopting naming practices into their curriculum; (2) we provide insights regarding the student perspective on naming practices, derived from the activities, revealing potential issues and opportunities in teaching the topic. [1]

Keywords

programming education
naming practices
course design
dialogic teaching
reflection
critical thinking
student perceptions

[1]Published as: van der Werf, V., F. Hermans, M. Specht, and E. Aivaloglou (2024). *Promoting Deliberate Naming Practices in Programming Education: A Set of Interactive Educational Activities*. In Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, **SIGCSE Virtual 2024**, page 235–241, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3649165.3690115

6.1 Introduction

From prior work, we know that variable naming practices are considered important by teachers, yet findings from introductory programming MOOCs, textbooks, and interviews with teachers [van der Werf et al., 2023, van der Werf et al., 2024c] **(Chapter 3, 4, 5)** indicate that many inconsistencies remain in teaching the subject matter. These inconsistencies are often due to variations in beliefs, goals, and intentions among teachers and designers of educational materials [van der Werf et al., 2024c] **(Chapter 3)**. Therefore it seems critical that the Computer Science Education community provides guidelines or approaches to teachers on handling naming practices.

However, any teaching approaches also need to consider the perspectives and attitudes of students regarding the topic. Such student perspectives largely remain unexplored and teachers seem to handle the topic based on personal experience: prior work showed teachers indicating that their students do not find naming a problem because they never receive questions about it [van der Werf et al., 2024c] **(Chapter 3)**. However, this could be a result of course design decisions or reflect factors such as the teacher's own beliefs or possible disinterest in the topic.

Since naming practices influence a programmer's code comprehension both positively and negatively [Hofmeister et al., 2017, Beniamini et al., 2017, Lawrie et al., 2007b, Lawrie et al., 2006, Feitelson, 2023, Schankin et al., 2018], we believe that software developers must have a thorough understanding of what makes a good name and be able to reflect critically on different naming practices. Our work presents a *dialogic teaching approach* to teaching a critical reflection on naming practices. We developed five types of activities that we ran and analyzed during a one-hour workshop given to a specialist vocational education program on software development. Through this workshop we were also able to explore students' perspectives and experiences on any barriers to adopting 'good naming practices', in this research denoted as names that carry the content or intent of the named object.

After presenting background on the topic of naming practices, we present our activities and their design (section 3), the workshop and its settings (section 4), and our experiences with the different activities (section 5). Finally, we reflect on our experiences and provide practical implications for the activity types.

6.2 Background

That (variable) naming is important for comprehension and code quality is indisputable from the existing literature focusing on the effect of naming on program comprehension, code quality, and coding skills. Most importantly, programmers rely on names for their understanding of code [Avidan and Feitelson, 2017, Hofmeister et al., 2017, Teasley, 1994, Takang et al., 1996, Lawrie et al., 2007b, Lawrie et al., 2006], and names often serve as beacons during code comprehension [Gellenbeck and Cook, 1991]. Moreover, bugs are easier to find when words are used [Hofmeister et al., 2017]. Additionally, names that are not descriptive enough, for example, single letters or abbreviations from which meaning is not directly clear, interfere with code comprehension [Lawrie et al., 2007b, Lawrie et al., 2006, Hofmeister et al., 2017, Beniamini et al., 2017]. The same holds true for too long names that can be difficult to remember [Binkley et al., 2009]. Additionally, names can

be unintentionally misleading and should therefore be chosen cautiously [Avidan and Feitelson, 2017, Arnaoudova et al., 2016, Feitelson, 2023, Feitelson et al., 2022]. Especially general, non-specific names, such as 'length' [Feitelson, 2023] or 'result' [Schankin et al., 2018], appear problematic. Finally, novices can wrongly believe that computers interpret or assign values based on the semantic meaning of variables' names, and thus incorrectly apply semantic assumptions to syntax [Kaczmarczyk et al., 2010].

Consequently, thinking about teaching variable naming in introductory programming courses becomes relevant. Thirty years ago, Keller [Keller, 1990] indicated that variable naming was rarely included in programming textbooks. Since then, little research observed teaching practices on this topic. Recently, Van der Werf et al. [van der Werf et al., 2023, van der Werf et al., 2024c] **(Chapter 3, 4, 5)** found that teachers addressed naming practices in their learning materials, but inconsistently: variable naming practices are not always taught explicitly, taught practices are sometimes conflicting, and given examples codes do not always match the provided rules and recommendations. About a decade ago, Glassman et al. [Glassman et al., 2015] developed a tool and a quiz for their online course (MOOC) to assess naming on length and vagueness. By evaluating the tool, they found that feedback on naming practices, as well as both good *and* bad examples, was highly valued by students. Unfortunately, no follow-up has been published since. Research investigating code quality perceptions among students and teachers [Börstler et al., 2017] confirmed students' desire for 'more and more specific feedback about what was good and bad in their code'. Other studies on variable naming in education found that novice programmers often fail to name variables correctly [Gobil et al., 2009] and that Scratch students are misled by variables named with a letter, probably because of prior knowledge from their mathematics education [Grover and Basu, 2017].

6.3 Activities - design & expectations

Since good naming practices depend on several factors, such as the context, programming language, purpose, and naming conventions, we argue that practitioners should focus on fostering a critical but adaptive attitude towards naming. Rather than teaching specific naming styles, our activities are designed to (1) strengthen students' reasoning about 'good' and 'bad' naming practices by encouraging them to reflect on names and (2) support an understanding of how names can influence code comprehension. To further support these objectives, we also focus on (3) creating awareness through personal experience by letting students explore their perceptions on the topic and making them experience various advantages, drawbacks, and limitations of different names for themselves. This, in turn, highlights the effects of naming choices. Finally, we aim to (4) train deliberate naming choices by building critical thinking skills applied to naming. This is crucial for in-depth reflection, especially knowing that students are expected to figure out naming 'by themselves' while feedback on naming is often missing [van der Werf et al., 2024c] **(Chapter 3)**.

Critical thinking, defined as 'reasonable reflective thinking focused on deciding what to believe or do' [Ennis, 2018], is often most effectively taught by combining *(critical) dialogue* with *authentic instruction* [Abrami et al., 2015]. Dialogue in this context covers learning through discussion, specifically including teacher-posed questions and teacher-led

whole-class discussion. Authentic instruction covers genuine and engaging problems such as applied problem-solving, case studies, simulations, games, and role-play. We maintain open-minded dialogue by applying the pedagogy of *dialogic teaching*, which is defined as 'a *general* pedagogical approach that embodies the strategic use of different types of talk, ranging from rote repetition to discussion, to achieve certain pedagogical goals' [Kim and Wilkinson, 2019]. Our activities therefore also centralize whole-class discussions and authentic examples.

In particular, we developed five different activity types, each with an opportunity to reflect usually through whole-class discussion and comparison of answers supported by an online polling system: (A) develop and express **perceptions**, experiences, and opinions, (B) **create** appropriate names for given variables within a code, (C) **rank** a set of given names based on (perceived) appropriateness or deceptiveness, (D) read and **compare** two identical codes with different names, and (E) **locate** a naming mistake in a code containing one misleading name. To stimulate reflection and discussion on naming, we facilitated program comprehension by *always* accompanying our code examples with a description of what the code does, its output, and the contents of each variable, both through the presented materials and the teacher. Below we discuss the activities separately before showing how we adapted them to develop a one-hour interactive workshop on naming and presenting our experiences per activity.

6.3.1 Activity Type A: Perceptions

This activity type stimulates students' reasoning and opinions on naming, encouraging them to develop and express their perceptions and own experiences. We designed two variants, one to "warm-up" (A1), focusing on activating and motivating students to explore the topic based on their prior experiences, and one to "wrap-up" (A2), aiming to consolidate opinions and establish students' viewpoints.

Variant A1 includes questions such as "when writing code, do you pay attention to naming?", "do you find naming an issue for software developers?", and "in your opinion, is naming worth the effort?". Students answer on a scale from 1 (never/not at all) to 10 (always/absolutely) through the online polling tool, which generates a summary of opinions to show the class as input for discussion. The teacher facilitates by prompting for more in-depth reasoning, and students are expected to participate by reacting to one another. Variant A2 asks students to individually write down their reasons for paying or not paying attention to naming practices and what prevents them from paying (more) attention to it. This can be implemented right after the discussion or at the end of the lesson. Alternatively, variant A2 could be given as preparation before class in a *flipped classroom* style with the intended interaction during class, serving the same purpose as A1.

6.3.2 Activity Type B: Create names

This activity uses student input on code snippets to lead the discussion, ensuring authentic instruction. The activity not only stimulates students to reason about appropriate names but also stimulates interest in the examples as the discussed names are their own. Additionally, the discussion allows for developing a common understanding of 'good' naming practices among the students.

Students are given a small code with redacted variable names. They individually name the redacted variables on paper and submit them anonymously through the online polling tool, which creates an overview of given names in the form of a list or word cloud. The teacher presents this overview, using it as the basis for discussion and prompting students to indicate what they notice about the set of names, which names they prefer and why, and what elements from these names they consider a part of "good" or "bad" naming.

6.3.3 Activity Type C: Evaluate through ranking

This activity encourages reflection by asking students to rank names for specific code snippets based on which they find most to least appropriate, or most to least misleading alternatively. By doing so, they rely on their perceptions and opinions to evaluate what they consider appropriate. Moreover, the activity provides an opportunity to experience that naming needs not be as straightforward as it seems at first sight. We expect different or opposing preferences to provide an ideal situation for discussion that is essential to reveal students' reasoning, show them how and why a name can be misleading to some, and help them understand the effect of names.

We designed two variants: (C1) ranking names from a given set per a single variable from most to least appropriate, and (C2) ranking names from a complete code snippet (each name representing a different variable) from most to least misleading. Again, a whole-class, teacher-led discussion is facilitated by submitting individual rankings to the online polling tool.

6.3.4 Activity Type D: Compare two codes

This activity type provides an opportunity to reflect on the effect of different naming styles on code comprehension by reading and comparing two codes only differing in the names representing the variables. By prompting students to compare the two codes and evaluate which they find easier to understand or more efficient, students further develop their perceptions. Moreover, by prompting students to reflect on which code looks more like those of other people and those written by themselves, they are stimulated to put the naming styles, including their own, in context.

We opted for two identical programs representing opposite naming styles: (1) letters and abbreviations, and (2) full word names. Students write down what they notice while comparing the codes and then select the program most fitting to four questions (which is easier, more efficient, looks like their programs, looks like other people's programs). They also explain their reasoning on paper.

6.3.5 Activity Type E: Locate the mistake

This activity aims for students to understand the effect of names on their understanding of code, showing them that names can be (unintentionally) deceiving and that choosing a good name might not be as straightforward as they might assume. We expect that this activity might serve as an 'eye-opener' to students when they struggle to identify the naming error. Discussion afterward is essential to reveal students' reasoning and to show them how and why a name can (sometimes) be misleading.

To mirror a real-life situation, we offer students code containing a (single) misleading name. To aid them, the explanation of the program, its output, and the contents of the variables are stressed (again). Students need to read and analyze the code to evaluate its names and are asked which name is wrong. Again, a whole-class, teacher-led discussion is facilitated by the online polling tool.

6.4 Workshop - design, setting & data

To test our activities, we developed a one-hour workshop covering all activity types (see **Figure 6.1**), and implemented it in March 2024 in two first-year classes within a three-year vocational program Software Development in an urban area in The Netherlands. We reached a total of 27 (male) students, aged 16-17. Twenty-one students gave consent to use their data. To accommodate the course, the first author visited the classes to do classroom observations, gaining a feel for the classroom interactions. We then developed the assignments in collaboration with the students' usual teacher, and presented the example codes in C#, as this is the language the students were learning, thereby eliminating possible confusion due to encountering an unfamiliar language. The first author led the workshop with the students' usual teacher present. The workshop was given in Dutch, including all the variable names used. Quotes and names presented in this chapter are all translated into English.

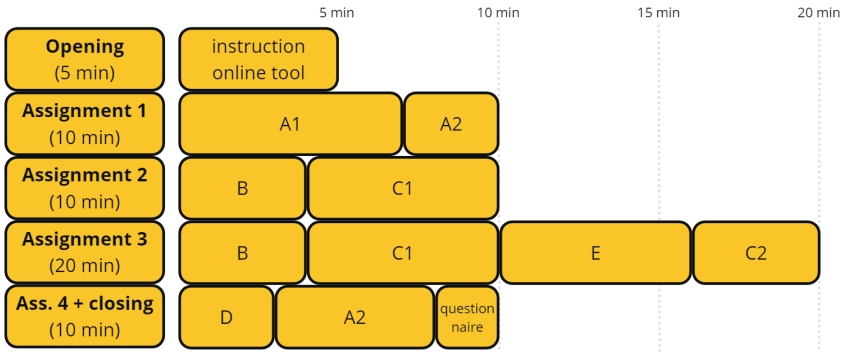


Figure 6.1: Overview of the workshop with time indications.

To collect data, we video recorded the front of the classroom, with only the whiteboard and the first author on tape, supported by additional audio recordings to capture students' verbal input observational notes of any events taken by a student assistant. The video recordings were transcribed and complemented with transcription from the audiotapes when necessary. Furthermore, we collected students' submissions in the online polling tool and their written contributions on paper hand-outs. These were digitized and added to the data from the polling tool using MS Excel. Finally, we collected students' experiences of the workshop through a questionnaire, part of the paper hand-out. The Ethics Review Committee of Leiden University approved this research.

It should be noted that students appreciated the online polling tool and it worked as intended. However, paper writing proved more challenging as students remarked it

had been "ages" since they had written anything with pen and paper. We also observed that the positioning of tables in the classroom influenced the workshop. There was a flexible seating arrangement with up to seven students per table. According to the teachers, this was the common setup, however, students were easily distracted by group dynamics within and between tables.

6.5 Workshop - experiences & results

6.5.1 Assignment 1: Activity Type A1 & A2

The 'warming-up' discussions (**Activity Type A1**) were successful in exploring students' perspectives. Students expressed they did not find naming an issue for software developers since "it is not that difficult at all." When asked whether they pay attention to naming, students responded diversely across the scale, giving ample room for discussion. Some students indicated that they gave specific attention because "it is important to keep track of your code," whereas others said they did not because "it is easy."

The written assignment directly afterward (**Activity Type A2**) showed more depth to students' reasoning: when prompted to name reasons for paying or not paying attention to naming, we found several themes, addressed in **Table 6.1**. Many students acknowledge that naming improves the code and benefits both understanding and writing, yet at the same time, they also express several issues. Most importantly, students indicate that paying attention to naming costs too much time and students fail to see its relevance. These results reveal that students have mixed experiences and opinions about whether or not naming deserves their attention, some of which hold them back from embracing the topic.

Table 6.1: Students' reasons for and against paying attention to naming and mentioned limitations while working with naming (obstacles).

Reasons for	Code clarity (11)
	Understanding (7): self (4), others (5), both (2)
	Code readability (4)
	Eases programming/debugging (3)
	Preventing errors (2)
Reasons against	Costs (too much) time (9)
	It is a non-issue (5): i.e., *"I already know"*,
	... *"it is not necessary"*, *"it does not matter"*
	Competes with writing/performance/accuracy (3)
	Costs too much effort (2)
Obstacles	Time (8)
	It is a non-issue (6): i.e., *"nonsense"*, *"unnecessary"*
	Possible confusion (3)
	Nothing (3)
	Too repetitive (1)

6.5.2 Assignment 2: Activity Type B and C1

To illustrate different naming options and their effects, we used the same code snippet for both activities, which converts temperature from Celsius to Fahrenheit (**Figure 6.2a**). After each activity, a whole-class discussion facilitated the understanding of how, why, and when the use of certain names can be counterproductive.

```
double A,B;
Console.WriteLine("Give temperature in Celsius");
A = Console.Read();
B = 1.8 * A + 32;
Console.WriteLine("Temperature in Fahrenheit:" + B);
```

(a) Convert temperatures from Celsius [A] to Fahrenheit [B].

```
double A,B,D;
int C;
Console.WriteLine("give [A]");
A = double.Parse(Console.ReadLine());
Console.WriteLine("give [B]");
B = double.Parse(Console.ReadLine());
Console.WriteLine("give [C]");
C = double.Parse(Console.ReadLine());
D = (A * B * C) / 100.0;
Console.WriteLine("Euro " + D.ToString("N2"));
```

(b) Calculate profit [D] from a savings account with [A] amount of money, [B] interest rate in %, for [C] number of years.

Figure 6.2: C# code snippets for assignments 2 (a) and 3 (b).

When asked to name variables [A] and [B] (**Activity Type B**), students predominantly write *celsius-fahrenheit* (8) or *tempCelsius-tempFahrenheit* (7), where "temp" could also be replaced by "degrees". Less popular were constructions like *temperature-fahrenheit* (3), *temperature-result* (1) and *number1-number2* (2). This demonstrates a preference for clarity, and perhaps already internalized conventions or community guidelines, but also reveals lazy and less informative attempts.

When ranking (best-worst) a set of name pairs for this code (**Activity Type C1**), we see a similar pattern (see **Figure 6.3**). However, students disagreed on the name pairs *temp1-temp2*, *input-output*, and *c-f*, which provided room for discussion in class: upon seeing the results of the ranking, students showed surprise, commenting, for example, that the name pair *c-f* was much better and more practical than *temperature1* or *temp1*, as these are too long and could be confusing. Although no student expressed it out loud, the disagreement for *temp1-temp2* could result from the common use of *temp* as an abbreviation for a temporary variable.

(n=18)	[A] celsius [B] fahrenheit	[A] degree [B] result	[A] temperature1 [B] temperature2	[A] temp1 [B] temp2	[A] input [B] output	[A] c [B] f	[A] a [B] b	[A] x [B] y
1st *[7 points]*	17	0	1	0	0	0	0	0
2nd	0	7	6	4	0	1	0	0
3rd	1	5	8	2	1	0	0	1
4th	0	4	2	7	2	3	0	0
5th	0	1	1	3	8	2	3	0
6th	0	1	0	0	4	3	7	3
7th	0	0	0	1	2	2	7	6
8th *[0 points]*	0	0	0	1	1	7	1	8
average rank	1,1	3,1	2,8	4,0	5,4	6,2	6,3	7,1
% of max points	98,4%	69,8%	74,6%	57,1%	37,3%	25,4%	23,8%	13,5%

Figure 6.3: Assignment 2, C1: Name-pair ranking

6.5.3 Assignment 3: Activity Type B, C1, E, and C2

Again, all activities use a single code snippet (see **Figure 6.2b**), allowing for experiencing and discussing the effect of different naming choices for a single code. The snippet presents a simplified calculation of the profit after saving a given amount for a given interest rate and a given time. Due to prior courses on the subject, students should be familiar with the economic context and terminology.

After explaining the program and each variable's contents, the students were asked to name the variables [A] to [D] (**Activity Type B**). Their answers revealed a preference for using a name that combines two words, such as *startingAmount*, *interestRate* or *numberOfYears* ("aantalJaar"). Also popular were single words such as *amount*, *balance*, or for variable [D], *result*, *outcome*, and *money*.

Interestingly, even though the variables' contents were discussed and provided, several students still made mistakes, writing *totalAmount* for variable [C] or *interest* for variable [D]. Moreover, while the names given to variables [A], [B], and [C] were mostly specific, the names given to variable [D] varied widely and showed little creativity. In fact, no student provided a name that included *profit*, which accurately describes the content. This could indicate a lack of domain knowledge. Alternatively, it could indicate an inability or unwillingness to translate the contents into a suitable name, or a lack of vocabulary or creativity. After seeing their classmates' answers, some students commented on the length of names by critiquing the combined names. This demonstrates a preference for shorter and more compact names.

For each variable, we presented students a set of names and asked them to rank from best to worst (**Activity Type C1**). Even though just before this activity, students clearly expressed they regarded combined names as "too long", the names students chose as "best name" overwhelmingly disregard that sentiment (see **Table 6.2**). Looking at the second, third, and fourth/last choice we mostly see clear 'winners' and shared 'losers'. For example, *amount*, *account*, and *start* received mixed positions, as did *years* and *term*, indicating they are considered equally bad, receiving very mixed positions.

Table 6.2: Students' 1st choice from a given list of names per variable (assignment 3: C1). The order shows the average ranking when a full ranking was provided (only 9/21 students).

[A] (n=18)	**[B]** (n=18)	**[C]** (n=21)	**[D]** (n=21)
startingAmount (17)	**interestRate (17)**	**numberOfYears (15)**	profit (5)
amount (0)	interest (1)	years (2)	result (5)
account (0)	percentage (0)	term (1)	outcome (2)
start (1)	perc (0)	time (0)	**endAmount (6)**
			calculation (0)

However, while patterns are more or less similar for variables [A], [B], and [C], students demonstrate different preferences for variable [D]: some students move toward *profit* as the best name, but *endAmount* and *result* remain equally popular. During the whole-class discussion, we witnessed students strongly defending their choices, although without clear and convincing arguments. When prompted what makes *calculation* so much worse than *outcome* or *result* the following discussion took place (translated):

S1: *"Calculation of what?"*
S2: *"It's too vague."*
S3: *"'Result' is more specific."*
T: *"But how about result/outcome 'of what'?"*
S1: *"Yes, but 'result' is much clearer."*
S3: *"It's the outcome of the calculation."*

One explanation for the preference for *result* might be that students are influenced by the "meta-program", where they prioritize the result of the function or program over a better reflection of the content. After all, the name *result* or *outcome* gives little information on what that result is composed of.

By locating a naming mistake (**Activity Type E**), students experience first-hand how certain naming practices can be unintentionally misleading. To illustrate this, we presented again the same code, but now with the names *startingAmount*, *interestRate*, *endAmount*, and *termInYears*, asking the students to find the mistake (endAmount).

Despite these efforts, this activity proved difficult for the students. Almost half of the students indicated they did not know, four students pointed to *interestRate*, another four to *termInYears*, and only three answered correctly. Since none of the students could explain why *endAmount* was misleading, the teacher attempted to make students get there by asking questions such as: "What does the variable represent?", "What does the name *endAmount* represent?", "What do you expect the program to deliver as output when the variable is called *endAmount*?", and "Does *endAmount* mean the same as *profit*?". Only with the last question, some students started to realize the mistake, but the majority still needed an explicit example. While the activity served as an eye-opener to many students, some continued to resist, commenting that "endAmount" still accurately represents the amount at the 'end' of the calculation. This further indicates that these students consider a certain 'meta-level' when choosing names, in a similar fashion as the names *result* and *outcome*, rather than choosing a name more indicative of its actual contents.

By having students rank all names from a code snippet from most to least misleading

(**Activity Type C2**), we create an authentic context in which students are stimulated to further explore how names can have negative effects. We prepared two different versions, each having a different set of (misleading) names. The most misleading names per version are presented in **Table 6.3** and show consistency with previous activities.

Table 6.3: Names chosen as most misleading (assignment 3: C2)

Variable	**Version 1** (n=19)	**Version 2** (n=11)
A	amount (3)	account (3)
B	percentage (5)	interestRate (1)
C	years (3)	**time** (7)
D	**calculation** (8)	result (0)

6.5.4 Assignment 4: Activity Type D and A2

By comparing identical codes with opposite naming styles (**Activity Type D**) students get another opportunity to experience the effect of naming choices. Students were unanimous in their opinion on which code they found easier (words) and most also indicated that this version looks more like their own programs. However, students were split in half on which approach was more efficient, showing that some prefer letters and abbreviations for efficiency, which is consistent with the opinions we found during Assignment 1 when students noted that paying attention to naming is too time-consuming and can compete with other objectives.

When prompted again with more general questions on naming practices (**Activity Type A2)**, we see that, compared to the start of the lesson, students are slightly more concerned. In more detail, we see four types of responses to the question *do you find naming a problem for software developers?* (scale 1-10). Those answering on the lowest end of the scale (1, never) say "it's not difficult" and "if naming would be a problem we have a big issue". Those rating 2-3 *and* those rating 7 and up note that naming "is no effort at all and helps immensely", whereas those rating 4-6 note that naming "costs a lot of time" and is a "big effort". Reactions to the question *do you find naming worth the effort?* can be found in **Figure 6.4**, which shows conflicting perspectives, especially among the middle group, and room for further dialogue.

6.5.5 Questionnaire (workshop evaluation)

Students rated the workshop with an average score of 7.6 out of 10 (n=21). Assignment Three was found most informative (n=8), followed by Assignment One (n=6) and Two (n=4). Half of the students (n=11) indicated that they plan to pay more attention to naming. These students also noted the following takeaways from the workshop: "naming is important," "the hows and whys of naming," and "to make deliberate decisions." On the other hand, eight students indicated that they do not plan to pay more attention to naming, even though their takeaways also included "names should be readable for another person too" and "always use correct names." These students may feel they already pay enough attention to naming, as one student also indicated "I didn't learn anything I didn't already know." Three students concluded the questionnaire by noting that they found the lesson enjoyable and very informative.

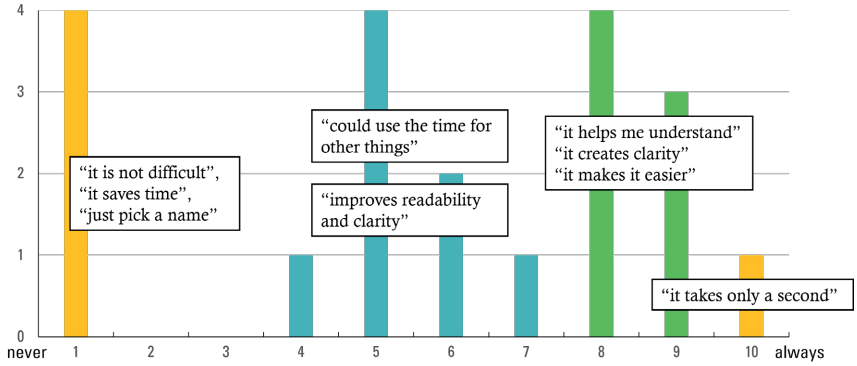


Figure 6.4: Results (n=19): is naming worth the effort?

6.6 Practical implications

Since naming practices influence code comprehension in various ways, but are currently taught inconsistently in introductory programming education [van der Werf et al., 2024c, van der Werf et al., 2023] **(Chapter 3, 4, 5)**, we experimented with an adaptive and interactive teaching approach to naming practices, supporting critical thinking, reflection, and deliberate naming choices. We presented five Activity Types and their design, discussed how they can be adapted for implementation, and showed our experiences with them. Below we highlight our insights and recommendations regarding the adoption of naming practices in a curriculum.

Use versatile activities Although we used C# code, the various activities support any programming language that uses labels for variables. Moreover, they support selecting and adapting existing programs from student code or examples from a running or previous course. This versatility helps create authentic experiences while limiting the time needed to develop new materials.

Encourage whole-class discussion Whole-class discussions proved extremely valuable in revealing potential issues (see below) and creating opportunities for reflection, nudging students to re-evaluate naming practices. We found the use of a third-party online polling tool beneficial in supporting the dialogue, but expect that other ways of promoting dialogue would have a similar effect, as long as students can share their choices and opinions with others and reflect upon those of others as well. Additionally, we have seen that reflection is deepened by also including activities that focus on individual consolidation of opinions after discussion.

Address and counter obstacles by offering various experiences Perception-focused activities (Type A) are successful in revealing potential issues for paying attention to naming practices, as students reported conflicting opinions and experiences. At least half of the students perceive paying attention to naming as (too) time-consuming, inefficient, and irrelevant, while also reporting finding the act of naming easy. If left unattended,

students could take a long time before experiencing the benefits of good naming choices. Considering that naming affects comprehension and heavily depends on many factors while teachers expect students to 'figure out naming by themselves' [van der Werf et al., 2024c] **(Chapter 3)**, a nonchalant attitude is unhelpful. Including a diverse set of activities provides (guided) opportunities for students to 'figure out naming' by practicing, experiencing, and reflecting on different naming styles, hence increasing the chances of gaining new perspectives. In this way, we can 'prime' students to adapt to a wide range of names and encourage the adoption of deliberate naming choices, without teaching specific naming styles.

Highlight the pitfalls surrounding naming At the same time, students already recognize reasons for paying attention, most often to improve code clarity and support code understanding for themselves and others. However, reflection based on ranking names (Activity Type C) and locating a naming mistake (Activity Type E) revealed that students are unaware of the limitations of certain naming choices and in particular of how naming choices can (unintentionally) deceive a reader. Through the activities, students showed increased awareness of how names are interpreted differently by different people, or when names, with the best intentions, do not accurately reflect the variable's contents. The teacher-led whole-class discussion was vital in increasing this awareness, highlighting both the strengths and weaknesses of naming choices. Our results show that using a single code snippet for various activities (Assignment Three) supports 'aha-moments' while using a variety of codes creates repeated practice with a wider range of examples.

6.7 Concluding remarks

In this work, we implemented a set of educational activities on variable naming in the single context of a vocational program with two small-size workshops totaling 27 (male) students. We stress that student participation, group dynamics, and teacher involvement all influence the outcomes of the discussions. However, we encourage practitioners to experiment with the activities also in other contexts, such as higher education or primary/secondary education. It would be especially interesting to compare these results with a similar workshop given to more experienced students, specifically related to our encountered (explicit) preference for names such as 'result', 'endAmount', and 'outcome'. We find this preference intriguing, as such names seem to be tailored to the 'meta-program' rather than the contents of the variable, and are known to be problematic for code comprehension [Feitelson, 2023, Schankin et al., 2018]. Finally, it could be interesting to use the activities with other programming languages, or even try out naming activities in a language the students are *not* familiar with, to see the effects on comprehension. Regarding comprehension, we also suggest experimenting with presenting code *without* an explanation of its purpose, its output, and/or a description of the contents of the variables. This likely increases difficulty and completion time, but would also train reading and program comprehension skills.

Chapter 7

Discussion and Conclusions

I started this dissertation with the quote: *"Miss, if naming already is an issue, we have a big probem!"*. My work revealed that the way this topic is currently taught in computer science education is inconsistent, and arguably, insufficient for the desired outcome of (1) professional programmers, or (2) high-quality code. My research has shown that addressing the topic of naming practices in education is important, and therefore, it deserves appropriate attention from the community.

The current chapter aims to provide several openings to support scholarly discussion within the community on how the topic can or should be addressed in the future. It also provides an opportunity for educators to reflect on their current practices and aims to support them in how they can address the topic in their courses to (better) match their learning philosophy and objectives.

The chapter is structured as follows:

- A recap of the aim and the research questions addressed in this dissertation;
- A discussion of this dissertation's key findings, organized by research question;
- A concluding summary which includes highlighted recommendations.

7.1 Recap of research aim and questions

This dissertation addressed programming education with the assumption that natural language serves as a bridge between complex programming problems and the programming language itself. Specifically, I aim to open a scholarly discussion on how naming practices can or should be implemented in programming education. The contributions to these objectives were laid out through the following chapters and research questions:

RQ1 *What do novice programmers express in their answers when asked to explain given code segments in their own words?* **(Chapter 2)**

RQ2 *How are variables and their naming practices introduced in beginner programming education and materials?* **(Chapters 3, 4, 5)**

RQ3 *What are teachers' beliefs and perceptions about naming practices and teaching them?* **(Chapter 3)**

RQ4 *How can we incorporate activities that focus on naming in beginner programming education?* **(Chapter 6)**

7.2 Key findings in context

In this section, I answer the research questions by discussing my key findings in context. Key findings are presented in line with the topics of the individual research questions.

7.2.1 Natural language in code and code explanations

Chapter 2 explored students' code explanations in plain English to answer: [**RQ1**] **What do novice programmers express in their answers when asked to explain given code segments in their own words?** This chapter provided insight into what novice students express in their explanations after reading a piece of code, and what these insights tell us about how the students comprehend code.

I found that novice programmers rely on the natural language present in the code when they are asked to explain a given code snippet. This reliance helps them interpret a code correctly but can also distract or misguide them into incorrect beliefs about the code's function, code constructs, or individual lines of code. My results also hint that some of these mistakes are instruction-related, meaning that with a change in the educational material, natural language-related mistakes and misconceptions could be avoided. On the other hand, adding natural language also revealed students' fragile understanding of programming constructs, which can be used by educators to address misconceptions.

Key Finding 1 *Natural language affects novices' program comprehension and potential learning.* **(Ch. 2)**

With this key finding, I underpin the relevance of natural language in programming. Often programming is associated with mathematics and problem-solving skills and it is not to say that such skills are not important; they are undeniably relevant to programming. However, the influence of natural language on code comprehension fits within a wider research context focused on natural language skills and strategies within the domain of programming education. Indeed, research that introduces natural language acquisition strategies in programming education appears promising. For example, reading code aloud helps to remember syntax [Hermans et al., 2018a, Swidan and Hermans, 2019] and vocabulary acquisition techniques help secondary education students in learning programming [Veldthuis and Hermans, 2024]. Moreover, there are even indications that training technical reading skills also improves programming skills [Endres et al., 2021a].

My findings also fit within a body of literature that advocates more structured programming courses with an eye for skill progression. These works move away from teaching programming skills through immediate code writing to solve mathematical problems and instead move toward course materials that also (first) focus on reading and understanding code structures [Xie et al., 2019, Sheard et al., 2014, Lopez et al., 2008, Venables et al., 2009]. As was highlighted in such works, skills such as code reading, code comprehension, and code tracing are good predictors of code writing skills. Hence practicing code writing before learning to read or trace code may increase the difficulty of becoming proficient in programming. This highlights that code reading and comprehension skills, affected by the natural language present in the code, are prerequisite skills and deserve more attention in programming courses.

Additionally, according to Schulte's Block Model [Schulte, 2008], understanding a program means being able to build a bridge from the lowest types of information and entity size (text:atom) towards the higher categories of either dimension (goals:macro) (see Figure 2.1 in Chapter 2). In other words, the process of code comprehension entails *translating*

the technical structures of a program to its social function. If indeed the natural language that is present in the code mediates the 'translation process', as my findings suggest, the academic community needs to further investigate how this finding can be used in educating novices with different backgrounds, including non-native English speakers.

My claim that natural language mediates comprehension can also be placed within the context that learning and motivation are affected when the used words or topic do not connect to the student's background. Indeed, research from the mathematics field is already familiar with the effect of language on so-called *story problems*. These are mathematical problems that require students to extract the relevant information from the story to solve a problem. Performance on such problems is affected by word choices and abstraction skills [Schley and Fujita, 2014, Mattarella-Micke and Beilock, 2010], meaning that elements such as names and objects (also known as *incidentals*) in the story can both help with comprehension and hinder it.

Some form of sensitivity to the words that are used, as I found in my study, was also found in computer science education research that opposed human-centered and thing-centered exercises as programming assignments [Christensen et al., 2021, Marcher et al., 2021]. These works found that problems focusing on humans rather than things are generally better understood and preferred by both women *and* men. Although my study did not investigate different student groups, it does further advocate for educators to be mindful regarding the topic and words they choose to use in example code. For example, I expect the effect of language to be bigger in unrepresented groups such as women, as objects can represent stereotypes of a group and deter those who do not identify with that group [Cheryan et al., 2009]. Hence, I invite researchers to dive into the potential differentiating effect of natural language in and around code on groups of minorities and varying levels of expertise.

On the other hand, completely obfuscating any natural language from a code limits the effect it has on comprehension. This could be especially useful to test how skilled a student is or has become in understanding 'pure' code structures. However, rather than testing programming skills, I am particularly questioning whether such obfuscation of any natural language from code is the most suited strategy for *teaching* novice learners, not in the least because within the profession, the use of natural language in code is often required to improve code quality and readability. Moreover, programming is hard to learn and students are already overwhelmed by the many new aspects involved with learning this new skill, so dismissing familiar elements likely increases difficulty for learning. Additionally, like with the mathematical story problems, abstraction skills –indisputably important for programmers– might be mediating the process. As such, training students to read code that includes natural language could facilitate the development of abstraction skills, while keeping unfamiliar elements to a minimum and the cognitive load limited. In this regard, future research could experiment with introducing (meaningful) natural language in code examples earlier or later on in the curriculum to investigate the effects on understanding new programming concepts and constructs as well as on abstraction skills. In this light, perhaps teaching students to structurally use comments, or introducing a more active practice of using sub-goal labels [Morrison et al., 2015] might be interesting too. The natural language that is already present in the code, through naming practices and input or output statements, may indeed mimic the effect of comments and sub-goal labels.

7.2.2 Variables and naming: current teaching practices

Chapter 3, Chapter 4, and Chapter 5 explore variable naming practices in particular and investigated teachers' perspectives, programming MOOCs, and programming books, to answer: [**RQ2**] **How are variables and their naming practices introduced in beginner programming education and materials?** These chapters painted a picture of the current landscape of how naming practices are taught, which served as the foundation for further investigation of how naming practices can and should be implemented. My observations revealed that educational materials and teachers (1) always give attention to the concept of variables relatively early in their course (materials) and (2) usually introduce naming practices along with the concept. Below I discuss my findings, for both topics separately.

Introducing the concept of variables

Generally, the concept of variables is introduced right after or just before data types and operators, although the order of introduction appears language-dependent. The concept is mostly described through the variable-as-a-box analogy, meaning that variables are typically explained as a box (or place) to store information in, often together with a visual representation. Python materials tend to include more diverse descriptions: they also introduce variables as a reference, name, or label. We observed that all materials focus their explanation on storing information, whereas other purposes such as keeping track of information, supporting code writing, interacting with information, or flexibility in using information, are rarely addressed. Only Scratch books mentioned that variables are called that way because their value can change. The common misconception that variables can store multiple values is rarely explicitly addressed.

Key Finding 2 *The introduction of the concept of variables is programming language-dependent. This is reflected by the chosen definitions and analogies and the position within the course compared to other programming constructs.* **(Ch. 4, 5)**

Key Finding 3 *The variable's purpose of storing information is extensively represented, whereas other functions or benefits of using variables are rarely mentioned, and potential misconceptions are rarely explicitly addressed.* **(Ch. 4, 5)**

Interpretations. These findings suggest that, while the concept of variables within the programming domain in itself is consistent across programming languages, the specific characteristics of these different languages may require a variety of approaches to teaching the concept, presumably because the use of variables may differ and their purpose might vary. However, while my findings give insights into current teaching practices, they cannot lead to definite conclusions about why they occur and therefore I stress that educational choices *seem* strongly influenced by the characteristics of the programming language that is taught. Whether this is the result of historical developments or the educator's preferences, it remains up to the community to decide whether it is desirable to adjust the teaching of general programming concepts to specific programming languages, especially

considering the many different languages that are in use nowadays. Moreover, as of yet, it remains untested if any of the approaches result in a stronger understanding of the concept, compared to others, and if that result is indeed language-dependent.

Implications. Hence, our findings have implications for possible 'universal' learning trajectories, and the effect of a programming language on understanding programming concepts and constructs should be further investigated. Much in the same line, our findings also have implications for the transfer of knowledge across programming languages. Using a variation of descriptions and including a multitude of purposes related to variables could help the understanding of the concept. However, we need to be mindful of students' cognitive load, hence more research is needed to understand which descriptions are most helpful: just adding more or alternative descriptions to the curriculum is not appropriate. Additionally, our results cannot exclude the effect of learner age and prior knowledge or extended skills and vocabulary obtained before their first programming lessons. While most adults might not need an explicit explanation of the 'changing' characteristic of variables, younger students or students who lack knowledge of English vocabulary may not yet be familiar with the word 'variable' in general, thus needing more explicit explanations. Yet, such an explicit connection within an explanation could prevent misconceptions also among adult learners as many programming languages use an equal sign (=) to assign values to variables, which does not immediately suggest that such a value can change.

Limitations. Since relatively few courses and textbooks were analyzed compared to the total offer of programming courses, my observations might not be complete. Nevertheless, the systematic analysis was purposefully performed on a wide range of popular courses and programming textbooks. Because similar results were obtained from these different educational materials, I am confident that the findings are representative. Moreover, our tip-of-the-iceberg overview now gives way for additional research to look into patterns between student age groups, student backgrounds, or the taught programming languages.

Recommendations. Based on Key Findings 2 and 3, I recommend that educators experiment with an extended range of definitions that include purposes beyond just 'storing information' and pay attention to the misconceptions that may arise from their chosen analogies. As was already hinted in prior research [Hermans et al., 2018b], different analogies have different effects on (young) learners and it is still unclear which type of explanations are most useful for long-term learning and transfer to other languages.

Introducing naming practices

The investigated programming courses and educational materials provide inconclusive, inconsistent, or even conflicting information regarding naming practices, and also teachers indicate that they do not pay attention to the topic. Materials predominantly focus on specific *syntax rules* that prevent the code from breaking, and formatting styles such as when to capitalize letters or use underscores. Materials and teachers also refer to various community guidelines that are often specific to a programming language and not directly provided to the students. Teachers and materials rarely discuss the topic more in-depth or reflect on how to name appropriately, what it means to name "meaningfully", or why naming practices are important. There are indications that naming is taught through a

constructivist pedagogical approach, in which students themselves discover by example what is good naming.

> **Key Finding 4** *Educators and teaching materials introduce naming practices inconsistently and they rarely address which, when, and why naming practices are (not) meaningful.* **(Ch. 3, 4, 5)**

This finding tells us that students are possibly insufficiently prompted to learn about the naming practices that are needed in their future careers. While a constructivist pedagogical teaching approach assumes that students learn from what they are confronted with through experiences and social interactions, any inconsistencies and discrepancies in (course) materials could prevent a coherent construction of knowledge. Indeed, the inconsistent teaching examples as observed in the course materials, programming textbooks, *and* as indicated by teachers, may (unwillingly) lead to the development of nonchalant attitudes toward naming practices. Instead of assisting student learning, students' current experiences with educational materials could thus hinder the adoption of a professional programmer's attitude. My finding therefore suggests that better attention needs to be paid to addressing and representing naming practices in programming education. It furthermore leaves an opening for adding a more explicit focus on *when* and *why* naming practices are important, which can be placed within the broader claim that programming education materials need to take a more structured approach. I propose that this is true whether or not a constructivist teaching philosophy is favored.

More implications and follow-up recommendations related to this key finding will be addressed in the next section, which addresses the educator's perspective. There I also demonstrate how the introduction of naming practices in course materials is embedded in the wider context of teachers' beliefs and assumptions, and what this means for students, teachers, and the academic community.

7.2.3 The educator's perspective: beliefs and strategies

Chapter 3 zoomed in on variable naming practices in particular and interviewed educators to answer: **[RQ3] What are teachers' beliefs and perceptions about naming practices and teaching them?** This chapter reveals several insights into how teachers think about naming practices and why they teach them the way they do. These insights serve as the foundation for further investigation of how naming practices can and should be implemented, explored later in the dissertation.

Teachers believe that names should be simple, straightforward, and intuitive, but there is no agreement on what this means in practice. As was already highlighted through **Key Finding 4**, this belief is not directly demonstrated in teaching approaches or materials. During the interviews, teachers mainly indicated to not explicitly incorporate the topic in their courses, nor encourage students to think critically about naming. They rarely grade naming practices or provide feedback to support students' self-reflection on their practices. Instead, they prioritize whether the student's code works and act from the assumption that naming is not difficult. The dominant philosophy is that naming practices are learned by example. At the same time, practical reasons prevent teachers from applying

their conviction of good naming practices in the examples they use in their educational materials, including exercises, slides, or live coding sessions. While this inconsistency happens across educational levels, it is more prominent in university teaching than in secondary education, where a more direct (instructive) approach is applied: some teachers developed and adopted more active strategies to support their students, and reflection on naming practices is repeatedly woven back into the curriculum through continuous feedback, specific (naming) assignments, and dedicated attention discussing repeated mistakes.

Key Finding 5 *Educators express that naming practices are important and that names should be 'meaningful'. However, most indicate not to pay explicit attention to the topic and do not require good naming practices from their students.* **(Ch. 3)**

Key Finding 6 *Educators assume naming practices are not difficult and are learned by example. However, they also indicate using examples that are inconsistent with their belief, for example out of practical reasons.* **(Ch. 3)**

Key Finding 7 *Educators do not require –nor wish to enforce– specific naming styles and they rarely encourage good naming practices through feedback.* **(Ch. 3)**

While all teachers stress the importance of naming practices for programming and that students need to become proficient at naming, my findings show that these beliefs are not evidenced in their teaching approaches. The lack of a persistent teaching approach is in line with the observations presented in **Key Finding 4**. This means that it is unlikely for educators to pay (much) explicit attention to naming practices, even though they consider them relevant.

This inconsistency could be explained by that teachers overestimate how important they find naming practices, especially when it is compared to other programming topics. Indeed, some university teachers expressed that other programming topics are more difficult and deserve more time and attention, hence downplaying the relevance of naming practices in programming education. It is also possible that educators wished to express "socially acceptable" opinions. This is a common problem in research on attitudes and opinions, although this is usually more prominent in research on socially sensitive or political topics such as discrimination or the protection of the environment. While I cannot exclude that such an effect may have played a role, I can say that several teachers reported that they realized their inconsistencies through their reflections during the interview and expressed a desire to correct them. Some secondary education teachers also expressed that they initially underestimated the complexity of the topic and learned to address the topic more explicitly through experience.

Rather than overestimating the importance of naming practices in programming, I argue that most educators perhaps underestimate the complexity or relevance of the topic for (novice) students. The topic competes with other programming topics for the limited

time in a curriculum and with teachers valuing these other topics as more important, naming practices end up drawing the short straw and are therefore (unintentionally) left out. My findings suggest that, even though naming practices are perceived as highly relevant within the community, most educators do not deem it necessary that explicit attention is given to the topic. However, my findings also suggest that this choice is usually made unconsciously, and more attention is given to the topic when teachers become more aware of the complexity of the topic for their students.

Nevertheless, the assumption that naming practices are not difficult and learned by example (**Key Finding 6**) supports the idea that the naming practices perhaps do not need explicit attention. However, to support learning when this assumption is true, the given examples should be consistent and match the teacher's philosophy on what is good naming. Instead, our findings suggest a mismatch between what students are expected to learn, and what educators are (unwillingly) teaching them. Rather than believing that educators do not care about the naming practices they teach their students, I argue that educators are generally not aware of the mismatch between their philosophy and practice.

Still, our finding that good naming practices are not required and rarely encouraged through feedback (**Key Finding 7**) also hints at the downplay of naming practices in programming education. Providing feedback on the topic might be considered unnecessary or too time-consuming, however, without any feedback, students lack the opportunity to check whether their naming can (or should) be improved. As a consequence, they may be led to believe that it is not important to use appropriate names in their code, let alone form a solid understanding of what good naming practices entail. Even when teachers tell their students that the topic is important, the lack of priority may suggest otherwise. Luckily, there are already initiatives that develop rubrics or tools to provide (large-scale) feedback on naming practices, and in a wider context, code quality. [Glassman et al., 2015, Börstler et al., 2017, Stegeman et al., 2014, Stegeman et al., 2016, van den Aker and Rahimi, 2024]. These can guide or inform teachers on how to incorporate feedback into their curriculum.

Implications for students. Unfortunately, it is yet unknown if the assumption that naming is learned by example is true. It is also unknown whether naming examples affect students of different ages differently. My research points to that students indeed copy examples they are shown in teaching materials and beyond, as teachers from secondary education especially highlight this. However, educators also warn that some of these students lack the understanding of why such names are chosen and copy them in inappropriate contexts. This begs the question of whether students should adopt a copy-paste strategy in the first place.

Teachers also relate 'bad names' with laziness or a lack of creativity on the student's part, rather than an inability to name appropriately. Based on my research findings (including **Key Finding 4**), it is possible that students never learned to name appropriately or do not care enough about it to pay attention to it themselves. This carelessness could reflect the inconsistent examples they were shown, which might lead students to believe that naming does not matter.

Implications for educators. Most importantly, educators and curriculum designers, including developers of educational materials such as MOOCs and books, need to be

aware of their philosophy about (teaching) naming practices and check if their practices reflect what they wish students to take away from their teaching. When teaching time is limited and the topic of naming practices competes with other programming concepts, it is essential that the little information that is (indirectly) given conveys a consistent and deliberate message to aid the student's learning process.

Furthermore, to encourage students' understanding of why certain names are (not) appropriate in which context, it might be wise to incorporate reflection on given examples as part of the teaching materials. This not only teaches them about naming practices and their importance but also encourages students to think critically and use their creativity. The lack of reflection and creativity, mentioned by teachers as holding students back from adopting good naming practices, could reflect the struggle that students experience while writing code. This may leave students (too) overwhelmed with other aspects of their learning process, and as a consequence, it limits their reflection and creativity. In light of this, I suggest training these skills outside of code writing activities. Instead, teachers could incorporate reading and reflection activities before or after writing activities that allow students to compare and reflect on written (example) code.

Implications for the academic community. Our findings furthermore mean that our academic community needs to investigate how different examples influence the learning process, and where in this process it is best to introduce more reflection on naming. For example, we do not know if it is (more) beneficial for learning new programming concepts and constructs if meaningful names are used in explanations and examples, or if random names (foo) or letters (a, s, x) are used. We also do not know if it is useful for students to spend (more) time reflecting on naming examples to improve their program comprehension skills. However, we do know that names influence program comprehension and that students who show better programming skills generally use better naming practices and vice versa. Moreover, code quality is considered important, therefore, even *if* naming skills do not improve overall programming skills, they should be learned to become proficient as a developer.

Limitations. We interviewed only a limited number of educators and deliberately included teachers from different educational levels and countries. This means that individual teacher perceptions may not be entirely representative, and future research should follow up with a large-scale (international) questionnaire to generalize and compare target audiences, class sizes, and class duration. This is interesting because my work indicates that the topic of naming is –and perhaps should be– addressed differently across educational levels and we lack the empirical insights into what are appropriate ways to teach the topic. Since we only interviewed teachers involved in Python programming courses, and there are indications that different programming languages are taught differently regarding the topic of (variable) naming practices, such larger-scale quantitative research could also investigate and compare a larger set of programming languages, which can reveal relevant results regarding potential transfer with the acquisition of a second programming language.

7.2.4 Implementing activities focused on naming

Chapter 6 aimed to inform educators on how to address naming practices in their course, based on the findings that were obtained through Chapters Two, Three, and Four. This foundation was used to design a set of interactive learning activities, addressing the research question [**RQ4**] **How can we incorporate activities that focus on naming in beginner programming education?** This chapter presented the value of discussions and dialogue in teaching the topic and shed light on several student issues that prevent a solid professional programmer's attitude toward naming.

Teacher-led whole-classroom discussions, centralized around various naming examples in presented code are useful in introducing the whats, hows, and whys of naming practices. Moreover, they can reveal issues among students that hinder the adoption of desired naming practices, namely the concern that paying attention to the topic is too time-consuming, inefficient, and even irrelevant. If we are to educate skilled professionals, these issues need to be actively attended to, especially in a teaching context where teachers believe that students will figure out naming by themselves.

The in-activity dialogues allowed for an increased awareness through repeated reflection on how chosen names can be (unintentionally) misleading to other readers. These dialogues are supported by using a single code snippet for various activities with changed variable names and by using a variety of codes to create repeated practice with a wider range of examples. This flexibility makes the activities versatile for use in courses with varying programming languages, levels, and other varying contexts. Moreover, the examples that were discussed provided opportunities for students to develop a sense of what belongs to 'good naming practices' in different contexts, and they can serve as a form of feedback on the topic. Because the activities were designed and implemented in the context of vocational education classrooms with 16-year-old students, it might be challenging to implement them in a large-scale university setting. However, I see an opportunity for scaling with the use of online tools and flipped classroom approaches as comparable prior work on social annotation in introductory programming courses shows positive results [De Oliveira Neto and Dobslaw, 2024].

Since naming practices are highly context-dependent and influence code comprehension in various ways, I recommend that educators focus their teaching on encouraging students to develop a critical attitude towards naming practices through reading and reflection exercises. This way, students learn to develop a grounded perspective on the topic and recognize potential issues. This also prevents passive copy-pasting strategies and moves away from teaching specific (language-dependent) styles that students might need to unlearn later on in their careers. Future research should look into how interactive teaching approaches (supporting critical thinking, reflection, and naming choices through dialogue) influence the understanding of programming constructs and code writing. Research should also investigate if such interactive activities can best be introduced early on in courses or could have a place in later courses. This research could also further explore why certain (unspecific) names such as 'result' and 'outcome' are favored by students and how the use of different types of names affects other programming skills. This can further inform the direction of in-class discussions.

7.3 Concluding summary

This dissertation focused on natural language (elements) in programming education and skills. I found that natural language can serve as a bridge between complex programming problems and the programming language itself **(RQ1; Chapter 2)**. Yet, programming education rarely teaches students how to use and interpret any natural language that is present in a code: specifically, my research found that students are rarely taught how to name their variables and functions and how such names can interfere with their code comprehension **(RQ2; Chapter 3, 4, 5)**. Educators believe that using good naming practices is an important skill for professional programmers, but assume learning this skill is not difficult and is generally done naturally by example **(RQ3: Chapter 3)**. Students are therefore not required to use good names and receive little to no feedback on their naming practices. Nevertheless, code examples used in courses and textbooks do not always reflect what teachers describe as good naming practices **(RQ2; Chapter 3, 4, 5)**, hence, the expectation that students learn by example might be compromised. Interactive activities that include whole-class dialogue based on various naming examples can raise awareness for the topic's importance, allow students to experience the effect of (unintentionally) misleading names, and provide opportunities for feedback needed to develop one's understanding of good naming practices **(RQ4; Chapter 6)**. Moreover, such activities revealed issues among students that may prevent the adoption of good naming practices.

Based on these results I make the following recommendations:

- *More awareness of the complexity of naming practices and their effect on learning programming skills is needed among educators and computing education researchers.* While educators and professionals agree on the importance of naming practices for professional developers and high-quality code, the topic seems to be overlooked in teaching programming skills.
- *More work and reflection is needed on whether and how programming education needs to actively teach skills on naming practices.* We already know that these practices are important for code comprehension, code quality, overall programming skills, and professional expectations, but know little about how these practices are acquired and how they may affect the adoption of other programming skills.
- *Academics need to further investigate the effect of naming practices –and in a wider context also natural language, code quality, and readability– on the adoption of programming skills.* This should ideally also contribute to a structured learning trajectory with an appropriate focus on other aspects of programming beyond problem-solving and code-writing abilities.
- *Educators need to be(come) aware of their philosophy on how they assume their students learn and adopt new skills and appropriately align their teaching approach.* Currently, there seem to be worrying inconsistencies between what is intended to be taught and what is taught in practice. If naming practices can be taught by example, examples should be consistent throughout educational material.
- *I encourage the adoption of interactive activities to explicitly address student issues, naming difficulties, and professional expectations.* These activities can be easily

adapted to varying contexts, programming languages, and presumably also classroom sizes, hence requiring relatively little effort for teachers, while providing authentic and repeated moments of reflection for students. Moreover, the focus on discussion and reflection opens up space for teaching naming practices beyond specific (language-specific) guidelines.

Bibliography

[Abrami et al., 2015] Abrami, P. C., Bernard, R. M., Borokhovski, E., Waddington, D. I., Wade, C. A., and Persson, T. (2015). Strategies for teaching students to think critically: A meta-analysis. *Review of Educational Research*, 85(2):275–314.

[Allamanis et al., 2014] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2014). Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA. Association for Computing Machinery.

[Antoniol et al., 2002] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983.

[Arnaoudova et al., 2016] Arnaoudova, V., Di Penta, M., and Antoniol, G. (2016). Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158.

[Avidan and Feitelson, 2017] Avidan, E. and Feitelson, D. G. (2017). Effects of variable names on comprehension: An empirical study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 55–65.

[Beniamini et al., 2017] Beniamini, G., Gingichashvili, S., Orbach, A. K., and Feitelson, D. G. (2017). Meaningful identifier names: The case of single-letter variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 45–54.

[Bettin and Ott, 2023] Bettin, B. and Ott, L. (2023). Pedagogical prisms: Toward domain isomorphic analogy design for relevance and engagement in computing education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 410–416, New York, NY, USA. Association for Computing Machinery.

[Bettin et al., 2023] Bettin, B., Ott, L., and Hiebel, J. (2023). More (sema|meta)phors: Additional perspectives on analogy use from concurrent programming students. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 166–172, New York, NY, USA. Association for Computing Machinery.

[Biggs and Collins, 1982] Biggs, J. B. and Collins, K. F. (1982). *Evaluating the Quality of Learning. The SOLO Taxonomy.* Elsevier Inc. Publication Title: Educational Psychology.

[Binkley et al., 2009] Binkley, D., Lawrie, D., Maex, S., and Morrell, C. (2009). Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445.

[Blinman and Cockburn, 2005] Blinman, S. and Cockburn, A. (2005). Program comprehension: Investigating the effects of naming style and documentation. In *AUIC*.

[Börstler et al., 2017] Börstler, J., Störrle, H., Toll, D., van Assema, J., Duran, R., Hooshangi, S., Jeuring, J., Keuning, H., Kleiner, C., and MacKellar, B. (2017). "i know it when i see it": Perceptions of code quality. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 389, New York, NY, USA. Association for Computing Machinery.

[Boulay, 1986] Boulay, B. D. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73.

[Brennan and Resnick, 2012] Brennan, K. and Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *American Educational Research Association*. ISSN: 1860949X.

[Briggs, 2023] Briggs, J. R. (2023). *Python for Kids: A plyaful introduction to programming (2nd edition)*. No Starch Press.

[Brooks, 1983] Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554.

[Busjahn and Schulte, 2013] Busjahn, T. and Schulte, C. (2013). The use of code reading in teaching programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, page 3–11, New York, NY, USA. Association for Computing Machinery.

[Butler, 2009] Butler, S. (2009). The effect of identifier naming on source code readability and quality. In *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ESEC/FSE Doctoral Symposium '09, page 33–34, New York, NY, USA. Association for Computing Machinery.

[Butler et al., 2015] Butler, S., Wermelinger, M., and Yu, Y. (2015). Investigating naming convention adherence in java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50.

[Butler et al., 2009] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2009). Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35.

[Butler et al., 2010] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2010). Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165.

[Caprile and Tonella, 1999] Caprile, B. and Tonella, P. (1999). Nomen est omen: analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 112–122.

[Caprile and Tonella, 2000] Caprile, B. and Tonella, P. (2000). Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*, pages 97–107, San Jose, CA, USA. IEEE.

[Cates et al., 2021] Cates, R., Yunik, N., and Feitelson, D. G. (2021). Does code structure affect comprehension? on using and naming intermediate variables. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 118–126.

[Charmaz, 2014] Charmaz, K. (2014). *Constructing grounded theory*. Introducing qualitative methods 181070847. SAGE, London, 2nd edition. edition.

[Chen et al., 2020] Chen, B., Azad, S., Haldar, R., West, M., and Zilles, C. (2020). A validated scoring rubric for explain-in-plain-english questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 563–569, New York, NY, USA. Association for Computing Machinery.

[Cheryan et al., 2009] Cheryan, S., Plaut, V. C., Davies, P. G., and Steele, C. M. (2009). Ambient belonging: how stereotypical cues impact gender participation in computer science. *Journal of personality and social psychology*, 97(6):1045–1060. Place: United States.

[Chiodini et al., 2021] Chiodini, L., Moreno Santos, I., Gallidabino, A., Tafliovich, A., Santos, A. L., and Hauswirth, M. (2021). A curated inventory of programming language misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 380–386, New York, NY, USA. Association for Computing Machinery.

[Christensen et al., 2021] Christensen, I. M., Marcher, M. H., Grabarczyk, P., Graversen, T., and Brabrand, C. (2021). Computing educational activities involving people rather than things appeal more to women (recruitment perspective). In *Proceedings of the 17th ACM Conference on International Computing Education Research*, ICER 2021, page 127–144, New York, NY, USA. Association for Computing Machinery.

[Clear et al., 2008] Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B., and Thompson, E. (2008). Reliably classifying novice programmer exam responses using the solo taxonomy. In Mann, S. and Lopez, M., editors, *Proceedings of the Twenty First Annual Conference of the National Advisory Committee on Computing Qualifications*, pages 23 – 30. National Advisory Committee on Computing Qualifications (NACCQ). 21st National Advisory Committee on Computing Qualifications, NACCQ 2008 ; Conference date: 04-07-2008 Through 07-07-2008.

[Corbin and Strauss, 2008] Corbin, J. and Strauss, A. (2008). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc., Thousand Oaks, California, 3rd edition.

[Corney et al., 2014] Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R., and Murphy, L. (2014). 'explain in plain english' questions revisited: Data structures problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, page 591–596, New York, NY, USA. Association for Computing Machinery.

[Corney et al., 2011] Corney, M., Lister, R., and Teague, D. (2011). Early relational reasoning and the novice programmer: Swapping as the "hello world" of relational reasoning. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, page 95–104, AUS. Australian Computer Society, Inc.

[Corney et al., 2012] Corney, M., Teague, D., Ahadi, A., and Lister, R. (2012). Some empirical results for neo-piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ACE '12, page 77–86, AUS. Australian Computer Society, Inc.

[De Oliveira Neto and Dobslaw, 2024] De Oliveira Neto, F. G. and Dobslaw, F. (2024). Building collaborative learning: Exploring social annotation in introductory programming. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 12–21.

[Deissenboeck and Pizka, 2006] Deissenboeck, F. and Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal*, 14(3):261–282.

[Doukakis et al., 2007] Doukakis, D., Grigoriadou, M., and Tsaganou, G. (2007). Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the 8th Hellenic European Research on Computer Mathematics & its Applications Conference, HERCMA'07*.

[Endres et al., 2021a] Endres, M., Fansher, M., Shah, P., and Weimer, W. (2021a). To read or to rotate? comparing the effects of technical reading training and spatial skills training on novice programming ability. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 754–766, New York, NY, USA. Association for Computing Machinery.

[Endres et al., 2021b] Endres, M., Karas, Z., Hu, X., Kovelman, I., and Weimer, W. (2021b). Relating reading, visualization, and coding for new programmers: A neuroimaging study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 600–612, Madrid, ES. IEEE.

[Ennis, 2018] Ennis, R. H. (2018). Critical Thinking Across the Curriculum: A Vision. *Topoi*, 37(1):165–184.

[Fakhoury et al., 2020] Fakhoury, S., Roy, D., Ma, Y., Arnaoudova, V., and Adesope, O. (2020). Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, 25(3):2140–2178.

[Feitelson, 2023] Feitelson, D. G. (2023). From code complexity metrics to program comprehension. *Commun. ACM*, 66(5):52–61.

[Feitelson et al., 2022] Feitelson, D. G., Mizrahi, A., Noy, N., Shabat, A. B., Eliyahu, O., and Sheffer, R. (2022). How developers choose names. *IEEE Transactions on Software Engineering*, 48(01):37–52.

[Fincher et al., 2020] Fincher, S., Jeuring, J., Miller, C. S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J. L., and Petersen, A. (2020). Notional machines in computing education: The education of attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '20, page 21–50, New York, NY, USA. Association for Computing Machinery.

[Fincher and Robins, 2019] Fincher, S. A. and Robins, A. V., editors (2019). *The Cambridge Handbook of Computing Education Research*. Cambridge Handbooks in Psychology. Cambridge University Press.

[Floyd et al., 2017] Floyd, B., Santander, T., and Weimer, W. (2017). Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186, Buenos Aires, Argentina. IEEE.

[Fowler et al., 2021] Fowler, M., Chen, B., Azad, S., West, M., and Zilles, C. (2021). Autograding "explain in plain english" questions using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, page 1163–1169, New York, NY, USA. Association for Computing Machinery.

[Gellenbeck and Cook, 1991] Gellenbeck, E. M. and Cook, C. R. (1991). An investigation of procedure and variable names as beacons during program comprehension. Technical report, Origon State University, USA.

[Gienow, 2017] Gienow, M. (2017). Code noob: The (variable) naming is the hardest part. https://thenewstack.io/code-noob-naming-hardest-part/

[Glassman et al., 2015] Glassman, E. L., Fischer, L., Scott, J., and Miller, R. C. (2015). Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, page 609–617, New York, NY, USA. Association for Computing Machinery.

[Gobil et al., 2009] Gobil, A. R. M., Shukor, Z., and Mohtar, I. A. (2009). Novice difficulties in selection structure. In *2009 International Conference on Electrical Engineering and Informatics*, volume 02, pages 351–356.

[Gresta et al., 2021] Gresta, R., Durelli, V., and Cirilo, E. (2021). Naming practices in java projects: An empirical study. In *XX Brazilian Symposium on Software Quality*, SBQS '21, New York, NY, USA. Association for Computing Machinery.

[Grover and Basu, 2017] Grover, S. and Basu, S. (2017). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 267–272, New York, NY, USA. Association for Computing Machinery.

[Hermans, 2020] Hermans, F. (2020). Hedy: A gradual language for programming education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 259–270, New York, NY, USA. Association for Computing Machinery.

[Hermans et al., 2018a] Hermans, F., Swidan, A., and Aivaloglou, E. (2018a). Code phonology: An exploration into the vocalization of code. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 308–311, New York, NY, USA. Association for Computing Machinery.

[Hermans et al., 2018b] Hermans, F., Swidan, A., Aivaloglou, E., and Smit, M. (2018b). Thinking out of the box: Comparing metaphors for variables in programming education. In *Proceedings of the 13th Workshop in Primary and Secondary Computing Education*, WiPSCE '18, New York, NY, USA. Association for Computing Machinery.

[Highland, 2019] Highland, M. (2019). *Coding for Kids: Scratch: Learn Coding Skills, Create 10 Fun Games, and Master Scratch*. ROCKRIDGE PR.

[Hofmeister et al., 2017] Hofmeister, J., Siegmund, J., and Holt, D. V. (2017). Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 217–227.

[Izu et al., 2019] Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C., and Weeda, R. (2019). Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 27–52, New York, NY, USA. Association for Computing Machinery.

[Kaczmarczyk et al., 2010] Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 107–111, New York, NY, USA. Association for Computing Machinery.

[Keller, 1990] Keller, D. (1990). A guide to natural naming. *ACM SIGPLAN Notices*, 25:95–102.

[Kennedy and Kraemer, 2019] Kennedy, C. and Kraemer, E. T. (2019). Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 224–230, New York, NY, USA. Association for Computing Machinery.

[Kenny and Fourie, 2015] Kenny, M. and Fourie, R. (2015). Contrasting classic, straussian, and constructivist grounded theory: Methodological and philosophical conflicts. *Qualitative Report*, 20(8):1270–1289.

[Keuning et al., 2019] Keuning, H., Heeren, B., and Jeuring, J. (2019). How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 119–125, New York, NY, USA. Association for Computing Machinery.

[Kim and Wilkinson, 2019] Kim, M.-Y. and Wilkinson, I. A. (2019). What is dialogic teaching? constructing, deconstructing, and reconstructing a pedagogy of classroom talk. *Learning, Culture and Social Interaction*, 21:70–86.

[Kirschner et al., 2006] Kirschner, P. A., Sweller, J., and Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, 41(2):75–86.

[Kohn, 2017] Kohn, T. (2017). Variable evaluation: An exploration of novice programmers' understanding and common misconceptions. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 345–350, New York, NY, USA. Association for Computing Machinery.

[Koksal, 2020] Koksal, I. (2020). The rise of online learning. https://www.forbes.com/sites/ilkerkoksal/2020/05/02/the-rise-of-online-learning/?sh=28f26e472f3c

[Laakso et al., 2008] Laakso, M.-J., Malmi, L., Korhonen, A., Rajala, T., Kaila, E., and Salakoski, T. (2008). Using roles of variables to enhance novices debugging work. *Journal of Information Technology Education: Innovations in Practice*, 5:281–296.

[Lawrie et al., 2007a] Lawrie, D., Feild, H., and Binkley, D. (2007a). Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388.

[Lawrie et al., 2006] Lawrie, D., Morrell, C., Feild, H., and Binkley, D. (2006). What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12.

[Lawrie et al., 2007b] Lawrie, D., Morrell, C., Feild, H., and Binkley, D. (2007b). Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318.

[Lehtinen et al., 2021a] Lehtinen, T., Lukkarinen, A., and Haaranen, L. (2021a). Students struggle to explain their own program code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 206–212, New York, NY, USA. Association for Computing Machinery.

[Lehtinen et al., 2021b] Lehtinen, T., Santos, A. L., and Sorva, J. (2021b). Let's ask students about their programs, automatically. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 467–475. Association for Computing Machinery.

[Lister, 2016] Lister, R. (2016). Toward a developmental epistemology of computer programming. In *ACM International Conference Proceeding Series*, volume 13-15-Octo, pages 5–16. Association for Computing Machinery.

[Lister, 2020] Lister, R. (2020). On the cognitive development of the novice programmer: And the development of a computing education researcher. In *Proceedings of the 9th Computer Science Education Research Conference*, New York, NY, USA. Association for Computing Machinery.

[Lister et al., 2009] Lister, R., Fidge, C., and Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, page 161–165, New York, NY, USA. Association for Computing Machinery.

[Lister et al., 2006] Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, page 118–122, New York, NY, USA. Association for Computing Machinery.

[Lopez et al., 2008] Lopez, M., Whalley, J., Robbins, P., and Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *ICER '08 - Proceedings of the ACM Workshop on International Computing Education Research*, pages 101–111. ACM Press.

[Makda and Mamazai, 2022] Makda, U. and Mamazai, T. (2022). *Python Coding for Kids Ages 10+*. Independently published.

[Marcher et al., 2021] Marcher, M. H., Christensen, I. M., Grabarczyk, P., Graversen, T., and Brabrand, C. (2021). Computing educational activities involving people rather than things appeal more to women (cs1 appeal perspective). In *Proceedings of the 17th ACM Conference on International Computing Education Research*, ICER 2021, page 145–156, New York, NY, USA. Association for Computing Machinery.

[Marji, 2014] Marji, M. (2014). *Learn to program with Scratch: A visual introduction to programming with games, art, science, and math*. No Starch Press.

[Mattarella-Micke and Beilock, 2010] Mattarella-Micke, A. and Beilock, S. L. (2010). Situating math word problems: The story matters. *Psychonomic Bulletin & Review*, 17(1):106–111.

[Matthes, 2023] Matthes, E. (2023). *Python Crash Course. A hands-on, project-based introduction to programming (3rd edition)*. No Starch Press.

[McMaster et al., 2016] McMaster, K., Rague, B., Sambasivam, S., and Wolthuis, S. (2016). Coverage of cs1 programming concepts in c++ and java textbooks. In *2016 IEEE Frontiers in Education Conference (FIE)*, page 1–8. IEEE Press.

[McMaster et al., 2018] McMaster, K., Rague, B., Sambasivam, S., and Wothuis, S. (2018). Software concepts emphasized in introductory programming textbooks. In *Proceedings of the 19th Annual SIG Conference on Information Technology Education*, SIGITE '18, page 91, New York, NY, USA. Association for Computing Machinery.

[Morrison et al., 2015] Morrison, B. B., Margulieux, L. E., and Guzdial, M. (2015). Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, page 21–29, New York, NY, USA. Association for Computing Machinery.

[Murphy et al., 2012a] Murphy, L., Fitzgerald, S., Lister, R., and McCauley, R. (2012a). Ability to 'explain in plain english' linked to proficiency in computer-based programming. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, page 111–118, New York, NY, USA. Association for Computing Machinery.

[Murphy et al., 2012b] Murphy, L., McCauley, R., and Fitzgerald, S. (2012b). 'explain in plain english' questions: Implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, page 385–390, New York, NY, USA. Association for Computing Machinery.

[Newman et al., 2020] Newman, C. D., AlSuhaibani, R. S., Decker, M. J., Peruma, A., Kaushik, D., Mkaouer, M. W., and Hill, E. (2020). On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740.

[Pelchen and Lister, 2019] Pelchen, T. and Lister, R. (2019). On the frequency of words used in answers to explain in plain english questions by novice programmers. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, page 11–20, New York, NY, USA. Association for Computing Machinery.

[Peruma et al., 2018] Peruma, A., Mkaouer, M. W., Decker, M. J., and Newman, C. D. (2018). An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, IWoR 2018, page 26–33, New York, NY, USA. Association for Computing Machinery.

[Prat et al., 2020] Prat, C. S., Madhyastha, T. M., Mottarella, M. J., and Kuo, C.-H. (2020). Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages. *Scientific Reports*, 10(1):3817.

[Rich et al., 2022] Rich, K. M., Franklin, D., Strickland, C., Isaacs, A., and Eatinger, D. (2022). A learning trajectory for variables based in computational thinking literature: Using levels of thinking to develop instruction. *Computer Science Education*, 32(2):213–234.

[Rich et al., 2017] Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., and Franklin, D. (2017). K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 182–190, New York, NY, USA. Association for Computing Machinery.

[Robbins, 2023] Robbins, P. (2023). *Python Programming for Beginners*. Independently published.

[Sajaniemi, 2002] Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 37–39.

[Sajaniemi and Kuittinen, 2005] Sajaniemi, J. and Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82.

[Sajaniemi and Prieto, 2005] Sajaniemi, J. and Prieto, R. N. (2005). Roles of Variables in Experts ' Programming Knowledge. In Romero, P., Good, J., Acosta Chaparro, E., and Bryant, S., editors, *17th Workshop of the Psychology of Programming Interest Group (PPIG17)*, pages 145–159. Sussex University.

[Salac and Franklin, 2020] Salac, J. and Franklin, D. (2020). If they build it, will they understand it? exploring the relationship between student code and performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 473–479, New York, NY, USA. Association for Computing Machinery.

[Salac et al., 2020] Salac, J., Jin, Q., Klain, Z., Turimella, S., White, M., and Franklin, D. (2020). Patterns in elementary-age student responses to personalized & generic code comprehension questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 514–520, New York, NY, USA. Association for Computing Machinery.

[Santos and Sousa, 2017] Santos, A. L. and Sousa, H. (2017). An exploratory study of how programming instructors illustrate variables and control flow. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling '17, page 173–177, New York, NY, USA. Association for Computing Machinery.

[Schankin et al., 2018] Schankin, A., Berger, A., Holt, D. V., Hofmeister, J. C., Riedel, T., and Beigl, M. (2018). Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 31–40, New York, NY, USA. Association for Computing Machinery.

[Schley and Fujita, 2014] Schley, D. R. and Fujita, K. (2014). Seeing the math in the story: On how abstraction promotes performance on mathematical word problems. *Social Psychological and Personality Science*, 5(8):953–961.

[Schulte, 2008] Schulte, C. (2008). Block model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 149–160, New York, NY, USA. Association for Computing Machinery.

[Sharif and Maletic, 2010] Sharif, B. and Maletic, J. I. (2010). An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 196–205.

[Sheard et al., 2014] Sheard, J., Simon, Dermoudy, J., D'Souza, D., Hu, M., and Parsons, D. (2014). Benchmarking a set of exam questions for introductory programming. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, pages 113–121. Australian Computer Society, Inc.

[Stegeman et al., 2014] Stegeman, M., Barendsen, E., and Smetsers, S. (2014). Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, page 99–108, New York, NY, USA. Association for Computing Machinery.

[Stegeman et al., 2016] Stegeman, M., Barendsen, E., and Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, page 160–164, New York, NY, USA. Association for Computing Machinery.

[Sweigart, 2020] Sweigart, A. (2020). *Automate the boring stuff with Python. Practical programming for total beginners (2nd edition)*. No Starch Press.

[Sweller, 2011] Sweller, J. (2011). *Cognitive Load Theory*, volume 55. Academic Press. ISSN: 00797421 Publication Title: Psychology of Learning and Motivation - Advances in Research and Theory.

[Swidan and Hermans, 2019] Swidan, A. and Hermans, F. (2019). The effect of reading code aloud on comprehension: An empirical study with school students. In *Proceedings of the ACM Conference on Global Computing Education*, CompEd '19, page 178–184, New York, NY, USA. Association for Computing Machinery.

[Swidan et al., 2017] Swidan, A., Serebrenik, A., and Hermans, F. (2017). How do scratch programmers name variables and procedures? In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 51–60.

[Tacke, 2019] Tacke, A. B. (2019). *Coding for Kids Python*. Rockridge Press.

[Takang et al., 1996] Takang, A. A., Grubb, P. A., and Macredie, R. D. (1996). The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Program. Lang.*, 4:143–167.

[Teasley, 1994] Teasley, B. E. (1994). The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies*, 40(5):757–770.

[Tshukudu et al., 2021] Tshukudu, E., Cutts, Q., Goletti, O., Swidan, A., and Hermans, F. (2021). Teachers' views and experiences on teaching second and subsequent programming languages. In *Proceedings of the 17th ACM Conference on International Computing Education Research*, ICER 2021, page 294–305, New York, NY, USA. Association for Computing Machinery.

[van den Aker and Rahimi, 2024] van den Aker, E. and Rahimi, E. (2024). Design principles for generating and presenting automated formative feedback on code quality using software metrics. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, ICSE-SEET '24, page 139–150, New York, NY, USA. Association for Computing Machinery.

[van der Werf, 2024] van der Werf, V. (2024). Fostering a natural language approach in programming education (doctoral consortium). In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, Koli Calling '23, New York, NY, USA. Association for Computing Machinery.

[van der Werf et al., 2022a] van der Werf, V., Aivaloglou, E., Hermans, F., and Specht, M. (2022a). (how) should variables and their naming be taught in novice programming education? In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2*, ICER '22, page 53–54, New York, NY, USA. Association for Computing Machinery.

[van der Werf et al., 2022b] van der Werf, V., Aivaloglou, E., Hermans, F., and Specht, M. (2022b). What does this python code do? an exploratory analysis of novice students' code explanations. In *Proceedings of the 10th Computer Science Education Research Conference*, CSERC '21, page 94–107, New York, NY, USA. Association for Computing Machinery.

[van der Werf et al., 2024a] van der Werf, V., Hermans, F., Specht, M., and Aivaloglou, E. (2024a). Promoting deliberate naming practices in programming education: A set of interactive educational activities. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*, SIGCSE Virtual 2024, page 235–241, New York, NY, USA. Association for Computing Machinery.

[van der Werf et al., 2024b] van der Werf, V., Hermans, F., Specht, M., and Aivaloglou, E. (2024b). Variables and variable naming in popular programming textbooks for children and novices. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*, SIGCSE Virtual 2024, page 242–248, New York, NY, USA. Association for Computing Machinery.

[van der Werf et al., 2024c] van der Werf, V., Swidan, A., Hermans, F., Specht, M., and Aivaloglou, E. (2024c). Teachers' beliefs and practices on the naming of variables in introductory python programming courses. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, ICSE-SEET '24, page 368–379, New York, NY, USA. Association for Computing Machinery.

[van der Werf et al., 2023] van der Werf, V., Zhang, M. Y., Aivaloglou, E., Hermans, F., and Specht, M. (2023). Variables in practice. an observation of teaching variables in introductory programming moocs. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 208–214, New York, NY, USA. Association for Computing Machinery.

[van Merriënboer and Kirschner, 2013] van Merriënboer, J. J. and Kirschner, P. A. (2013). *Ten steps to complex learning. A systematic approach to four-component instructional design.* Routledge, second edition.

[Veldthuis and Hermans, 2024] Veldthuis, M. and Hermans, F. (2024). A word about programming: Applying a natural language vocabulary acquisition model to programming education. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E '24, page 56–65, New York, NY, USA. Association for Computing Machinery.

[Venables et al., 2009] Venables, A., Tan, G., and Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *ICER '09 - Proceedings of the 2009 ACM Workshop on International Computing Education Research*, pages 117–128. ACM Press.

[Vermeulen et al., 2000] Vermeulen, A., Ambler, S. W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., and Thompson, P. (2000). *The Elements of Java Style*. Cambridge University Press.

[Vorderman et al., 2017] Vorderman, C., Steele, C., Quigley, C., Goodfellow, M., McCafferty, D., and Woodcock, J. (2017). *Coding Projects in Python*. DK Publishing.

[Vorderman et al., 2018] Vorderman, C., Steele, C., Quigley, C., McCafferty, D., and Goodfellow, M. (2018). *Coding Games in Python*. DK Publishing.

[Waguespack, 1989] Waguespack, L. J. (1989). Visual metaphors for teaching programming concepts. *SIGCSE Bull.*, 21(1):141–145.

[Wainewright, 2020] Wainewright, M. (2020). *Code Your Own Games!: 20 Games to Create with Scratch*. Union Square Kids.

[Weeda et al., 2020] Weeda, R., Izu, C., Kallia, M., and Barendsen, E. (2020). Towards an assessment rubric for eipe tasks in secondary education: Identifying quality indicators and descriptors. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, Koli Calling '20, New York, NY, USA. Association for Computing Machinery.

[Whalley and Kasto, 2014] Whalley, J. and Kasto, N. (2014). How difficult are novice code writing tasks? a software metrics approach. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, pages 105–112. Australian Computer Society, Inc.

[Whalley et al., 2006] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., and Prasad, C. (2006). An australasian study of reading and comprehension skills

in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, page 243–252, AUS. Australian Computer Society, Inc.

[Woodcock, 2015] Woodcock, J. (2015). *Coding Games in Scratch: A Step-by-Step Visual Guide to Building Your Own Computer Games*. Penguin.

[Woodcock, 2016] Woodcock, J. (2016). *Coding Projects in Scratch: A Step-by-Step Visual Guide to Coding Your Own Animations, Games, Simulations, and More!* Penguin.

[Xia et al., 2018] Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., and Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976.

[Xie et al., 2019] Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., and Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3):205–253.

Afterword and acknowledgments

Had anyone told my 2019-past self that I was going to pursue a PhD at the Leiden Institute of Advanced Computer Science, I would have never believed them. While my career path has been anything but straightforward, the leap from archaeology to educational sciences felt like a natural one (what's more embedded in society if not education?), and the leap to data analytics was a logical one (what's data if not pieces of information to interpret, like archaeology?). By transitioning to computer science education, I aimed to contribute to the education of the future. Although I am likely the only one walking this unique path, I have crossed paths with many people to whom I owe thanks.

To start, I thank my supervision team. First, I owe my gratitude to *Felienne*, for your passion, drive, and inspiration; for without it, I would not have been here. I still vividly remember our first conversation after seeing one of your animated talks. "Be brave. You never know what a first meeting might lead to." (James Norbury, Big Panda and Tiny Dragon). I would also not have been here without *Fenia*; you always knew like clockwork when I needed a little encouragement. Your 'complaint' of "I want to use my red pen and draw arrows and stuff," may have been the single most helpful comment throughout the entire process. And of course, I am grateful for your warm advice, our rejection drinks, and your home dinners, which made me feel part of an academic family. I also thank *Marcus*, you blindly trusted me to make it happen, and from you I learned to believe in myself. I thank all of you for the opportunity to work at LIACS and CEL, and for bringing together a bunch of 'lost' PhD students.

I extend my gratitude to my colleagues and peers at Leiden and Delft University, with whom I had much-needed coffee breaks, tears, and laughter, and with whom I shared hardship, networks, food, and advice. Thank you for the opportunities you gave me to grow and experiment. Special thanks to (no particular order): *Martine Baars*, for trusting me to teach in your course, which I then shamelessly used to experiment according to my own philosophy: dialogue instead of PowerPoint. For me, that was a powerful milestone. *Gillian Saunders-Smits*, for your unwavering support and advice as a woman in science; and for your academic guidance, introducing me to your network, and supporting and enabling the work of our own. *Gitte*, for being my partner in crime as colleagues and friends. The endless flow of fantastical ideas, your fanaticism, our covid homework sessions, our adventures –on and off conferences–, and so much more, have made my journey memorable. I cannot imagine anymore NOT sharing our joy, sadness, frustration, and academic enthusiasm. I am truly grateful to have you as my friend. *Marcella*, for our immediate connection; you too, I have very quickly considered friends first, colleagues second. You've 'hooked' me into crochet and shown me so much strength, endurance, and enthusiasm. Please continue to show everyone the way to being a beautiful, smart, and ambitious superwoman. *Johannes*, for walking up to a few strangers at the hotel lobby and fully embracing our company, crazy stories, and ambitious ideas; you have made our collaboration an absolute joy, bringing in your pragmatism, enthusiasm, planning, and good questions. I would not hesitate to work together again. *Shirley*, for making it look easy to walk the path you lay out for yourself and not letting anyone get to you. And behind your calm front, there is a ton of creativity, of which I can only hope you keep it. *Xiaoling*, for your unwavering dedication to work and hobbies, and your zest for learning everything that finds itself on your path. *Erna*, for your dry humor and for never skipping lunch. *Nesse*, for our long and enthusiastic talks and walks that reminded me doing a PhD is also fun. *Manuel*, for always being in for social drinks and always knowing the right (AI) tools and prompts, both in work and fun settings (who said they are mutually exclusive?) #DrankmakerijDeKievit.

I furthermore thank the SEFI, ITiCSE, and Koli communities, whose members embraced my 'otherness' and encouraged their PhDs. Special thanks to *Barbara Ericson, Matt Daniels, Nick*

Faulkner, Mark Guzdial, for mentoring us towards confidence in the best Doctoral Consortium I have been, also due to my peer students (*Dimitri, Jesse, Sebastian, Katrin*) and the biggest bag of Kinderbueno. I also thank *Lauri Malmi* for the lovely conversation we had on educational theory; even though I only dared to talk to you at the very end of the evening, you were a great reminder of the beautiful academic discourse that can happen just by meeting someone with a similar passion.

My friends outside academia, you kept reminding me, with or without intention, of life outside the university. A special thanks to those who have made an impact on who I am during my PhD journey; I have so much respect for who you are! *Willemijn*, for being your brave, spontaneous self, for puzzles and tea and talks and drinks; *Dirk*, for always being a challenging opponent in almost any game and truly believing in me as if it is the single most normal thing to do; *Mara*, for your awesome energy, strong and reliable advice, and being someone I always look forward to talking to again and again; *Rosanne*, for making our 15-year old friendship feel unchanged even when we are not in contact; and *Kayley*, for reaching out and being so open and inviting. I also thank *Fenny, Dorien, Cailin & Ian, Rosa, Caitlin, Frederic, Jeff* and *Eva* for your unique contributions to my life these last years.

Finally, I am immensely thankful for my close family, who are always there for me, trusting in me, and giving advice. I have much respect for my mom, *Willeke*, who finished her PhD right before I started mine; you were in many ways my rock, and your often spot-on advice is as good as internalized. Thank you for raising me in a surrounding that exposed me to lots of cultures, interests, and opinions. Likewise, I can always rely on the practical and pragmatic advice from my dad, *Willy*; your voice in my head keeps reminding me to work to live, not the other way around. I am lucky to feel how proud you both are of me. *Marijn* and *Fré*, I think you both are so smart and brave and cool, and even though I wish I could be more like you, I could never do what you do. And finally, *Coen*, now pursuing your own PhD, my 30th would not have been the same without you!

I strongly believe all events are connected, as I can connect each choice I made leading to the next cool thing I got to do. In a more general sense, I am proud to say that I have become a sort of human time machine, able to see and connect with the past (archaeology), the now (education), and the future (computer science). All three have become a huge part of who I am and how I view the world. I owe my gratitude to all of you who contributed to this, including my teachers, mentors, professors, and fellow students along the way.

Summary

Learning a programming language is challenging. Even one of the first –but crucial– concepts to learn, variables, is hard to grasp. Variables can represent different roles and functions within a code and need to be named appropriately to support code comprehension and debugging. Yet, students can hold various misconceptions about variables that can be (unintentionally) encouraged by the explanations and analogies used to introduce and explain the concept. Moreover, also names can be (unintentionally) misleading, hindering the understanding of the concept and the code.

This dissertation approaches the teaching of a programming language from the reasoning that (familiar) natural language, like the language used to name variables, serves as a bridge between complex programming problems and the programming language itself. The dissertation first investigates students' code explanations in plain English, after which it delves deeper into the teaching of variables and (variable) naming practices in introductory programming education across educational levels. These include secondary education, vocational education, university education, adult education, massive open online courses (MOOCs), and programming textbooks for children and novices.

Throughout the chapters, this dissertation aims (1) to open a scholarly discussion on how naming practices can or should be implemented in programming education, and (2) to inform and support educators and developers of educational materials in the fields of *Computer Science* and *Software Engineering* who want to know how to address the topic in their courses and materials. The performed research is mostly qualitative and exploratory: it involves an analysis of student artifacts (chapter 2), open coding of in-depth interviews with teachers on their perceptions and practices (chapter 3), and observations of educational courses and materials (chapters 4 and 5). However, the research ends with the design and implementation of interactive learning activities to support the adoption of good naming practices among learners (chapter 6).

In short, the dissertation presents the following results. Novice programmers rely on natural language elements that are present in code, to understand and explain it (chapter 2). Yet, students are rarely taught how to name their variables and functions nor how names can interfere with code comprehension (chapters 3, 4, 5). While educators believe that using good naming practices is an important skill for professional programmers, they assume learning this skill is not difficult and is generally done 'naturally' and 'by example' (chapter 3). As such, students are seldom required to use appropriate names and receive little to no feedback on their naming practices (chapter 3), which limits their opportunities to learn from mistakes. Moreover, code examples used in courses and textbooks do not always reflect what is described as good naming practices by teachers *and* those same (course) materials (chapters 3, 4, 5). Finally, the introduction of the concept of variables appears programming-language-dependent, and the used explanations and analogies show a dominant focus on storing information; other functions or benefits of using variables are rarely mentioned and potential misconceptions appear not addressed (chapters 4, 5).

These results show that many students are expected to learn from conflicting information without much support or guidance. Hence, the expectation that students learn by example could be compromised and naming practices deserve more careful and explicit attention in programming courses. The designed interactive activities (chapter 6) include whole-class dialogue based on various naming examples which can raise awareness for the topic's importance, allow students to experience the effect of (unintentionally) misleading names, and provide opportunities for feedback needed to develop one's understanding of good naming practices. Moreover, such activities revealed issues among students that may prevent the adoption of good naming practices, such as rejecting the topic's importance and cost-benefit-related issues (chapter 6).

For educators, the main takeaways are: (1) pay more explicit attention to the importance of developing good naming practices; (2) use consistent naming examples in regular example codes; (3) integrate interactive activities to discuss *how* and *why* names are appropriate or inappropriate; (4) stimulate students' critical thinking on the effect of naming choices; and (5) provide regular feedback on students' naming choices. Furthermore, educators are encouraged to (6) expand the definitions and analogies they use for introducing the concept of a variable in their teaching to better reflect variables' purpose and use in programming and address potential misconceptions.

Lastly, this dissertation challenges the academic community to further investigate the effect of naming practices on students' *learning process* and the adoption of (future) programming skills. Ideally, such research should provide guidelines and contribute to a structured learning trajectory with an appropriate focus on aspects of programming that go beyond problem-solving and code-writing abilities and include more reading and (interactive) comprehension activities.

Samenvatting

Leren programmeren is moeilijk. Zelfs een van de eerste, en misschien wel een van de belangrijkste concepten van een programmeertaal, variabelen, is ingewikkeld. Variabelen kunnen verschillende rollen en functies aannemen binnen een code en moeten de juiste naam krijgen om het begrip van code en andere programmeeractiviteiten zoals debuggen te ondersteunen. Leerlingen kunnen echter verschillende misvattingen hebben over variabelen, waarvan sommige zelfs (onbedoeld) kunnen worden aangemoedigd door de uitleg en analogieën die worden gebruikt wanneer het concept wordt geintroduceerd. Bovendien kunnen ook namen zelf ook (onbedoeld) misleidend zijn, wat het begrip van zowel het concept en de code belemmert.

Dit proefschrift benadert het onderwijzen van een programmeertaal vanuit de redenering dat (bekende) natuurlijke taal, zoals de taal die wordt gebruikt voor de naamgeving van variabelen, als brug dient tussen complexe programmeerproblemen en de programmeertaal zelf. Het onderzoek behandelt eerst de geschreven uitleg van code in de eigen woorden van studenten, en gaat daarna dieper in op het onderwijzen van variabelen en naamgeving in programmeeronderwijs voor beginners van alle leeftijden (waaronder voortgezet onderwijs, beroepsonderwijs, universitair onderwijs, volwassenenonderwijs, *massive open online courses* (MOOC's) en leerboeken over programmeren voor kinderen en beginners).

Het proefschrift beoogt (1) een wetenschappelijke discussie te openen over hoe naamgevingspraktijken kunnen of zouden moeten worden geïmplementeerd in het programmeeronderwijs, en (2) docenten en ontwikkelaars van onderwijsmateriaal op het gebied van computerwetenschappen en softwaretechniek te inspireren, informeren, en te onder-steunen over hoe het onderwerp in cursussen en materialen behandeld kan worden. De uitgevoerde onderzoeken zijn kwalitatief en verkennend van aard en omvatten een analyse van studentenmateriaal (hoofdstuk 2), open codering van diepte-interviews met docenten over hun visie en onderwijspraktijken met betrekking tot naamgeving (hoofdstuk 3), en observaties van onderwijscursussen en -materialen (hoofdstukken 4 en 5). Het laatste onderzoek (hoofdstuk 6) presenteert interactieve leeractiviteiten die zijn ontworpen op basis van de uitkomsten van de eerdere studies. Deze activiteiten zijn getest en dienen de adoptie van goede naamgevingspraktijken onder lerenden.

Kort samengevat worden de volgende resultaten gepresenteerd. Beginnende programmeurs vertrouwen op natuurlijke taalelementen die aanwezig zijn in code, om deze te begrijpen en uit te leggen (hoofdstuk 2). Toch wordt studenten zelden geleerd hoe ze hun variabelen en functies een naam moeten geven en hoe namen het begrip van code kunnen belemmeren (hoofdstuk 3, 4, 5). Hoewel docenten geloven dat het gebruik van goede naamgevingspraktijken een belangrijke vaardigheid is voor professionele programmeurs, gaan ze ervan uit dat het aanleren van deze vaardigheid niet moeilijk is en over het algemeen 'natuurlijk' en 'via voorbeelden' gebeurt (hoofdstuk 3). Als zodanig worden studenten zelden verplicht om de juiste namen te gebruiken en krijgen ze weinig tot geen feedback op hun naamgevingspraktijken (hoofdstuk 3), wat hun mogelijkheden om van fouten te leren beperkt. Bovendien weerspiegelen de voorbeeldcodes die in cursussen en leerboeken worden gebruikt niet altijd wat als goede naamgevingspraktijken wordt beschreven door docenten in diezelfde (cursus)materialen (hoofdstuk 3, 4, 5). Tot slot lijkt de introductie van het concept variabelen afhankelijk van de programmeertaal en de gebruikte uitleg en analogieën focussen zich voornamelijk op het opslaan van informatie; andere functies of voordelen van het gebruik van variabelen worden zelden genoemd en mogelijke misvat-tingen lijken niet aan de orde te komen (hoofdstukken 4, 5).

Deze resultaten laten zien dat van studenten wordt verwacht dat ze leren van tegen-strijdige informatie zonder veel ondersteuning of begeleiding. De verwachting dat men leert van voorbeelden is daarmee niet zonder compromis en naamgevingspraktijken verdienen zorgvuldige en expliciete

aandacht in programmeercursussen. De ontworpen activiteiten (hoofdstuk 6) zijn interactief en richten zich op een klassikale dialoog op basis van verschillende naamgevingsvoorbeelden. Hiermee pogen de activiteiten het bewust-zijn over het belang van naamgeving te vergroten, laten ze leerlingen het effect van (on-bedoeld) misleidende namen ervaren, en bieden ze veel ruimte voor de nu vaak missende feedback die nodig is voor verdere ontwikkeling van goede naamgevingsvaardigheden. Bovendien brachten de activiteiten problemen aan het licht die de adoptie van goede naamgevingspraktijken in de weg kunnen staan, zoals een algehele afwijzing van het belang van het onderwerp en specifieke kosten-batengerelateerde kwesties (hoofdstuk 6).

De belangrijkste conclusies voor docenten zijn: (1) besteed meer expliciete aandacht aan het belang van het ontwikkelen van goede naamgevingspraktijken; (2) gebruik consistente naamgevingsvoorbeelden in regelmatige voorbeeldcodes; (3) integreer interactieve activiteiten om te bespreken hoe en waarom namen gepast of ongepast zijn; (4) stimuleer kritisch denken van leerlingen over het effect van naamgevingskeuzes; en (5) geef regelmatig feedback op de naamgevingskeuzes van leerlingen. Verder worden docenten aangemoedigd om (6) de definities en analogieën die ze gebruiken om het concept van een variabele te introduceren in hun onderwijs uit te breiden om het doel en het gebruik van variabelen in programmeren beter weer te geven en mogelijke misvattingen aan te pakken.

Tot slot daagt dit proefschrift de academische gemeenschap uit om verder onderzoek te doen naar het effect van naamgevingspraktijken op het leerproces van studenten en de ontwikkeling van (toekomstige) programmeervaardigheden. Idealiter draagt dit bij aan gestructureerd leertrajecten die zich (ook) focussen op aspecten van programmeren die verder gaan dan vaardigheden voor probleemoplossing en het schrijven van code, zoals bijvoorbeeld het lezen van code, kritisch nadenken over- en reflecteren op gemaakte keuzes.

Curriculum Vitae

Vivian van der Werf was born in Tilburg, The Netherlands, in 1992. She completed her Bachelor's degree (BA) in Archaeology in 2013 and her Master's degree (MA) in Archaeology in 2018, both at Leiden University. Meanwhile, she pursued a second Master's degree by completing her pre-master's in Child and Education Studies in 2014 and her Master's degree (MSc) in Educational Sciences in 2018, both at Leiden University. During this time, she was a member of the Educational Committee at the Faculty of Archaeology for two years and completed the International Leiden Leadership Program of the Honours Academy Leiden in 2017. She has worked for about two and a half years as a data analyst and policy officer before starting her PhD research in 2020 at the Leiden Institute of Advanced Computer Science. Vivian guest lectured upon invitation at the Erasmus School of Social and Behavioral Sciences in 2022 and 2023 with Dr. Martine Baars, gave workshops on implementing peer feedback (national and international), job crafting, and variable naming in programming, and assisted in the bachelor's course Studying and Presenting in 2023. She furthermore took courses in academic resilience, qualitative research, and scientific conduct, among others. Currently, Vivian is obtaining her BVNT2 certificate at the Vrije Universiteit Amsterdam and works as a Dutch language teacher for international students, expats, and immigrants, with a special focus on women and minorities who are (re)skilling themselves for the Dutch tech/IT sector.

List of publications

** Also published in this dissertation.*

- van Helden, G., V. VAN DER WERF, J. Schleiss, G.N. Saunders-Smits (2025), *FIETS: A Tool for Assessing the Embedding of Theory in Engineering Education Intervention Research*. In **European Journal of Engineering Education**. doi: 10.1080/03043797.2025.2532589
- *****VAN DER WERF, V., F. Hermans, M. Specht, and E. Aivaloglou (2024). *Promoting Deliberate Naming Practices in Programming Education: A Set of Interactive Educational Activities*. In Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, **SIGCSE Virtual 2024**, page 235–241, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3649165.3690115
- *****VAN DER WERF, V., F. Hermans, M. Specht, and E. Aivaloglou (2024). *Variables and Variable Naming in Popular Programming Textbooks for Children and Novices.* In Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, **SIGCSE Virtual 2024**, page 242–248, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3649165.3690112
- *****VAN DER WERF, V., A. Swidan, F. Hermans, M. Specht, and E. Aivaloglou (2024) *Teachers' Beliefs and Practices on the Naming of Variables in Introductory Python Programming Courses.* In 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), **ICSE-SEET 2024**, page 368–379, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3639474.3640069
- VAN DER WERF, V. *Fostering a natural language approach in programming education (doctoral consortium).* In Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, **Koli Calling 2023**, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3631802.3631838
- VAN DER WERF, V., G. van Helden, J. Schleiss, G.N. Saunders-Smits (2023). *A Framework for Investigating Educational Theories in Engineering Education Research.* European Society for Engineering Education (**SEFI 2023**). doi: 10.21427/PM7V-MD26
- *****VAN DER WERF, V., M.Y. Zhang, E. Aivaloglou, F. Hermans, and M. Specht (2023). *Variables in Practice. An Observation of Variables in Introductory Programming MOOCs*. In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, **ITiCSE 2023**, page 208–214, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3587102.3588857
- van der Meer, N., V. VAN DER WERF, W.P. Brinkman, M.M. Specht (2023). *Virtual reality and collaborative learning: A systematic literature review.* In **Frontiers in Virtual Reality**, vol. 4. doi: 10.3389/frvir.2023.1159905
- van Helden, G., V. VAN DER WERF, G.N. Saunders-Smits, M.M. Specht (2023). *The Use of Digital Peer Assessment in Higher Education—An Umbrella Review of Literature.* In **IEEE Access**, vol. 11, 22948-22960. doi: 10.1109/ACCESS.2023.3252914
- G.N. Saunders-Smits, G. van Helden, V. VAN DER WERF, M.M. Specht (2022). *Using peer assessment in inclusive digital education (workshop)*. European Society for Engineering Education (**SEFI 2022**).

- VAN DER WERF, V., E. Aivaloglou, F. Hermans, and M. Specht (2022). *(How) Should Variables and Their Naming Be Taught in Novice Programming Education?* In Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2, **ICER 2022**, page 53–54, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3501709.3544288
- *VAN DER WERF, V., E. Aivaloglou, F. Hermans, and M. Specht (2021). *What does this Python code do? An exploratory analysis of novice students' code explanations.* In Proceedings of the 10th Computer Science Education Research Conference, **CSERC 2021**, page 94–107, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3507923.3507956