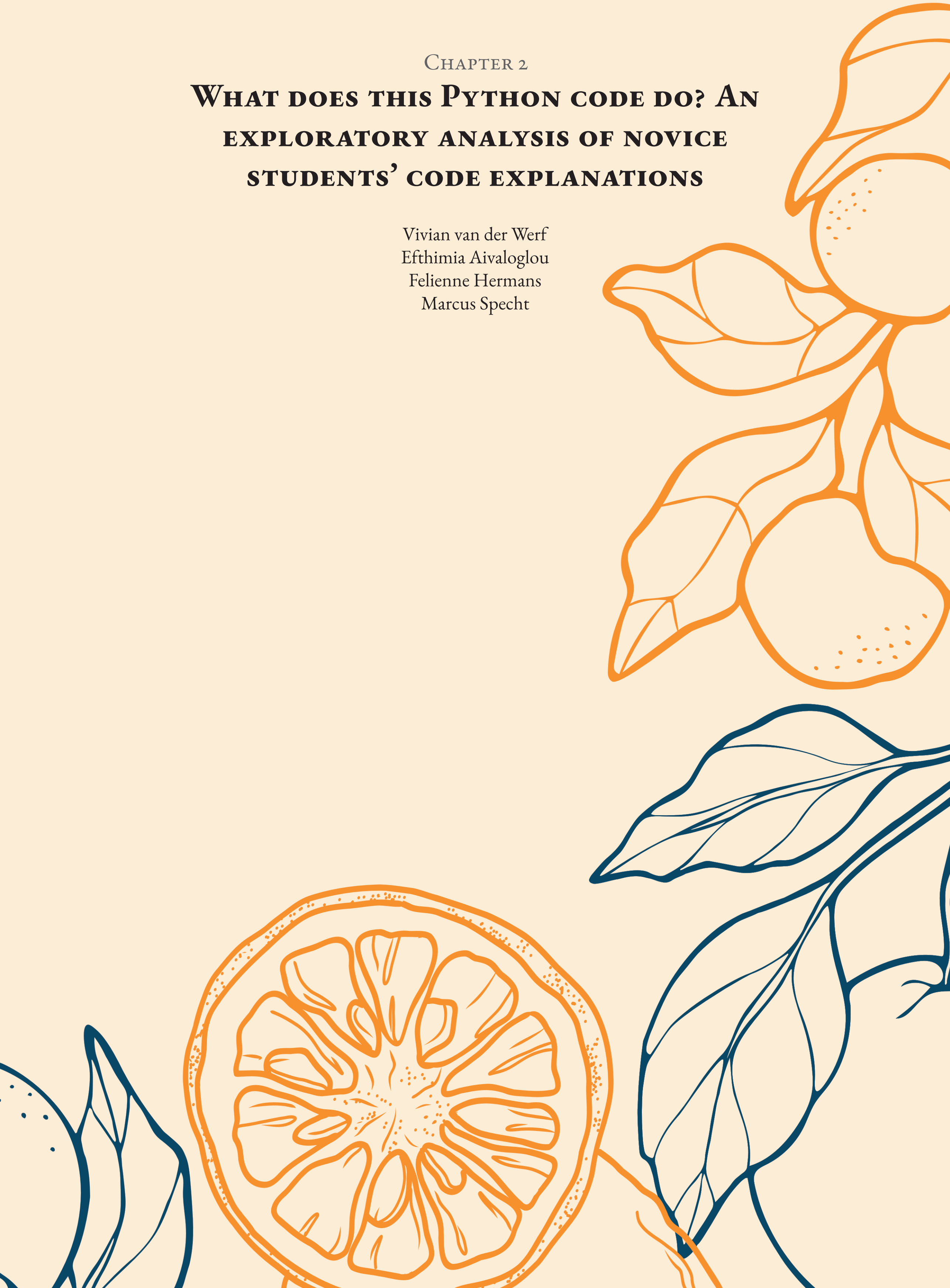
Chapter 2

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

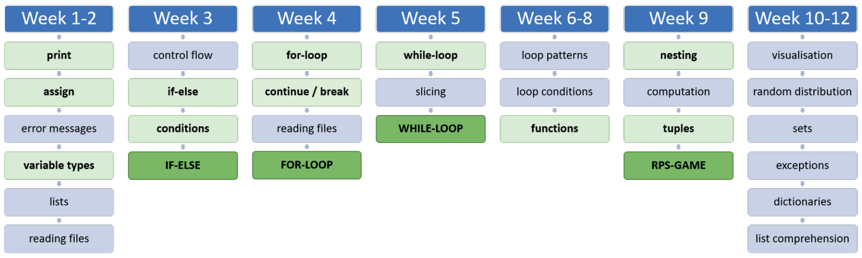
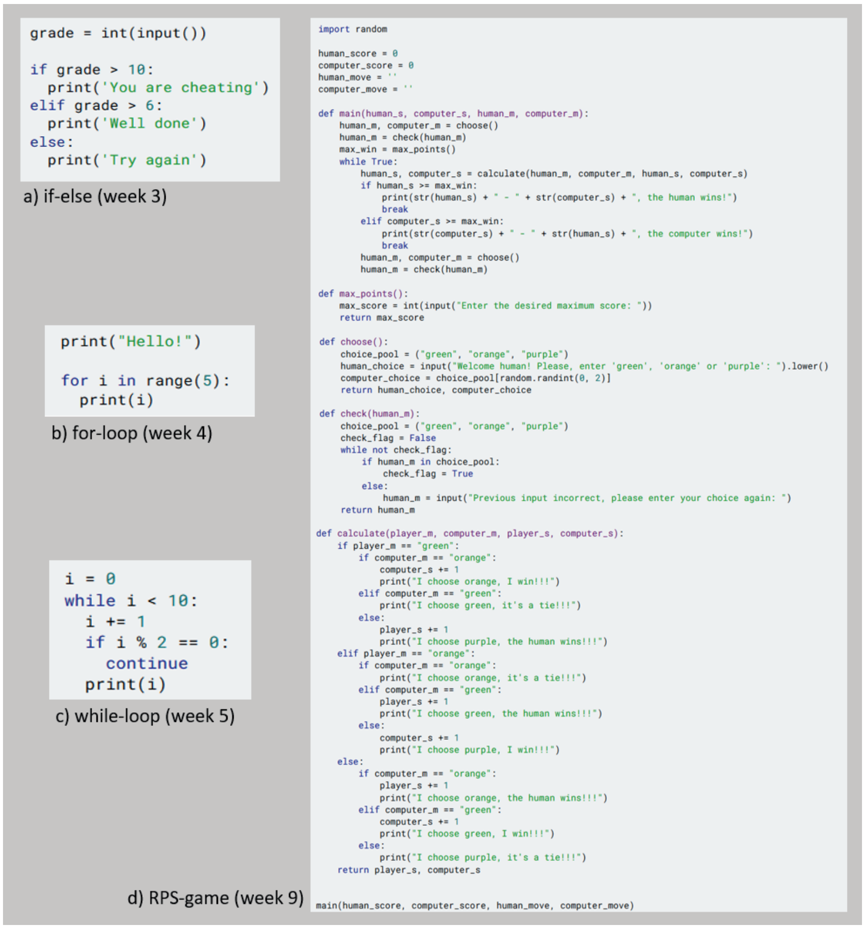


Figure 2.1: Course structure with the programming concepts covered in the course. Light green: first introduction of the programming concepts; dark green: indication of when the reading exercises analyzed in this chapter were introduced.

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 assignstatements 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	A	B	C	D	Total	A	B	C	D
TOTAL N	66	12	15	16	23	%	%	%	%	%
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.