

From Generation to Education: Integrating Biology into Programming Exercises

Gino Van Cleemput

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science option Artificial
Intelligence

Supervisor

Prof. dr. ir. T. Schrijvers

Assessors

dr. ir. Van Roy

Prof. dr. S. Nijssen

Assistant supervisor

Ir. T. Fernanda Ugarte Guzmán

© 2027 KU Leuven – Faculty of Engineering Science

Published by Gino Van Cleemput

Department of Computer Science Celestijnenlaan 200A bus 2402 B-3001 Leuven

All rights reserved. No part of the publication may be reproduced in any form by print photoprint microfilm electronic or any other means without written permission from the publisher. This publication contains the study work of a student in the context of the academic training and assessment. After this assessment no correction of the study work took place.

Preface

cknowledgment I would first like to sincerely thank my promoter and supervisor for their guidance, support, and mentorship throughout this thesis. Their feedback and advice were extremely valuable, and I learned a great deal from working with them during this project.

I would also like to thank the students from the educational master's programme for their collaboration and contributions to this thesis. Working together gave me valuable insights into education and the pedagogical perspective behind the design of the programming exercises.

special thank you goes to my girlfriend for her patience, understanding, and continued support throughout the thesis process, especially when I spent more time working on this thesis than cycling during our biking holiday.

Finally, I would like to thank my family for reading parts of this thesis, providing feedback, and taking on as much of the other workload as possible so that I could focus on finishing it.

Gino Van Cleemput

Contents

Preface	i
Abstract	iv
List of Figures and Tables	v
List of abbreviations and symbols	viii
1 Educational context and problem setting	1
2 Literature Review	3
2.1 Theoretical frameworks for motivation and learning experience . . .	3
2.2 Pedagogical foundations for contextualized programming exercises .	9
2.3 Automatic generation of educational programming exercises	16
3 Materials and methods	25
3.1 Study design and scope	25
3.2 Exploratory phase: design context	27
3.3 From design context to system requirements	28
3.4 Prompt development methodology	32
3.5 System architecture	33
3.6 Internal system evaluation (phase 3)	36
3.7 Human-centered evaluation (phase 4)	38
4 Results and discussion	43
4.1 Findings from the exploratory study	43
4.2 Iterative development of the generation architecture	46
4.3 Internal behavior of the exercise-generation layer (<i>Phase 3</i>)	52
4.4 General generation architecture (<i>Phase 3</i>)	57
4.5 Teacher and domain-expert evaluation <i>Phase 4</i>	61
4.6 Classroom study	63
4.7 Conclusion	66
4.8 Future work	69
Exploratory Study Instruments	73
4.1 Teacher Survey	73
4.2 Semi-Structured Interview Guide	86
B Classroom Study Questionnaire Instruments	91
B.1 Pre	91

B.2	Post	95
C	Teacher and Domain-Expert Evaluation Instruments	103
C.1	Teacher Questionnaire	103
C.2	Domain-Expert Questionnaire	114
C.3	Variable exercise sequence	128
C.4	Conditionals exercise sequence	147
C.5	While loop exercise sequence	167
C.6	Teacher Follow-Up Interview Guide	189
D	Supplementary Results of the exploratory interviews and teachers survey	193
D.1	Supplementary Teacher Survey Results	193
D.2	Exploratory Interview Findings	194
E	eTeacher representation	197
E.1	Relevant programming-exercise fields	197
E.2	Test representation used in the generated exercises	198
F	Examples from the Design based System Development	199
F.1	Baseline and Modular Generation Examples	200
F.2	Modular generation example	203
F.3	Feedback-Based Refinement Examples	205
F.4	Dedicated Test-Generation Example	208
F.5	eTeacher Y ML output and platform specific test cases	209
F.6	External review notes	210
G	Detailed Consistency Evaluation Results	215
G.1	Teacher Blueprints	215
G.2	Generated blueprint	215
H	Feedback- gent Evaluation	217
H.1	Quantitative Results	217
H.2	Representative Feedback- gent Refinements	220
I	Test Execution and Refinement Behavior	225
I.1	Execution and Refinement Metrics	225
I.2	Illustrative Refinement Examples	227
J	Supplementary Results of the Teacher and Domain-Expert Evaluation	231
J.1	Teacher evaluation	231
J.2	Domain-expert evaluation	232
	Bibliography	235
	Code of Conduct	239

bstract

This thesis investigates how large language models can support the creation of interdisciplinary programming exercises for students in Biological and Chemical STEM Sciences, with biology providing a meaningful context for learning introductory programming.

Using an exploratory design-based research approach, the study develops and evaluates a generation architecture that combines structured educational input, contextual grounding, generation, evaluation, refinement, and platform-specific output. The architecture was first implemented for individual exercise generation and subsequently applied to sequence generation

The resulting system and generated material were evaluated from three complementary perspectives. Internal evaluation examined the behavior and reliability of the generation pipeline, teachers and domain experts assessed the quality and classroom usability of generated sequences, and students evaluated their experience with the contextualized exercises. The findings show that the architecture provides a promising basis for controlled interdisciplinary exercise generation while preserving a role for teacher adaptation and control. Remaining limitations include test generation, diversity of exercise types, and information presentation. Students generally perceived the biological context as making the usefulness of programming more visible, although no clear general increase in motivation was observed.

List of Figures and Tables

List of Figures

2.1	The self-determination continuum,[7]	5
3.1	Overview of the four study phases, the iterative development of the exercise-generation layer, and their relationship to the research questions.	25
3.2	Prompt design process used for creating agents and prompt sections. The process starts from a defined goal, translates this goal into output requirements and prompt structure, and uses output evaluation to refine the corresponding prompt or instruction when needed.	32
3.3	Generation workflows used for the classroom study and the teacher and domain-expert evaluation.	38
4.1		44
4.2	General architecture	58
4.3	The results on how students experienced the contextualized exercises	64
D.1	Results of the teacher survey on the perceived suitability of progressively structured exercise sequences across broad programming categories.	193
F.1	What a LLM can generate without much guidance	200
F.2	Limitation: Not python test cases	200
F.3	JSON output	201
F.4	Python test cases	202
F.5	Limitation: to many features	202
F.6	Enter Caption	203
F.7	Enter Caption	203
F.8	Reference to a previous exercise	204
F.9	Incorrect terminology	204
F.10	Wrong phrasing	205
F.11	Teacher agent flags issue	205
F.12	Teachers flagged issue is successfully resolved	205
F.13	Student agent flags issue	206
F.14	Students flagged issue is successfully resolved	206

LIST OF FIGURES AND TABLES

F.15 Problem in test case generation	207
F.16 Identified test case issue	208
F.17 How this iteration solved that issue	208
F.18 Generates meaningful feedback when test fails	209
F.19 Direct mapping to eTeacher format	209
F.20 Successful test generation but bad feedback generation	210
H.1 inconsistency was flagged by both the students and the teacher agent	220
H.2 Enter Caption	220
H.3 Not the correct terminology used	221
H.4 Correcting terminology mistake in template and solution code	221
H.5 Correcting terminology mistake in problem description	222
H.6 Instruction in the problem statement and scaffold are not aligned	222
H.7 The instruction in the problem statement were adjusted	222
H.8 This wording could lead to confusion	223
H.9 The wording was improved	223
I.1 No lines were printed	227
I.2 Solved: Source was a mismatch between test generation and the execution environment	227
I.3 simple syntax error was identified	227
I.4 There remained an error: mismatch between execution environment and the test generation	228
I.5 The solution code was not executed	228
I.6 The problem was solved by removing the lines without solving the issue	228
I.7 Enter Caption	229

List of Tables

3.1 Inputs to the generation engine and the resulting output artifacts.	29
3.2 Functional requirements of the generation engine and their design rationale.	30
3.3 Evaluation framework used to assess the quality and teacher usability of generated exercises and sequences.	31
3.4 Exercise-experience categories used to organize the post-intervention questionnaire items for the contextualized group.	41
4.1 Overview of the architectural iterations and their resulting capabilities.	46
4.2 Observed failure patterns in refined executions.	57
4.3 Comparison of how the general architecture is implemented in the exercise- and sequence-generation layers.	60
D.1 Conditions for successful implementation of interdisciplinary programming exercises identified in the open survey responses. P1 did not provide a response to this question.	194

D.2	Summary of the main themes identified across the three exploratory interviews. Interview findings are based on written notes and are therefore presented as paraphrased themes rather than direct quotations.	195
E.1	Main eTeacher fields relevant to the generated programming exercises.	197
E.2	Relevant fields of the eTeacher output-test representation used in the generated exercises.	198
G.1	Pairwise correlations between consistency metrics for teacher-defined exercise Blueprints.	215
G.2	Mean pairwise similarity scores per teacher-defined exercise blueprint across 30 generation runs.	215
G.3	Mean pairwise similarity scores per system-generated exercise blueprint across 30 generation runs.	216
G.4	Pairwise correlations between consistency metrics for system-generated exercise blueprint.	216
H.1	Feedback-agent verdicts and component changes for teacher-defined and system-generated specifications. Each specification was processed 30 times. “Teacher agent only,” “student agent only,” and “both agents” indicate which feedback perspectives requested revision. Component-change counts indicate runs in which the draft and final component were not identical. (abbreviations: T = teacher-defined specification; G = system-generated specification.)	218
H.2	Final similarity metrics for teacher-defined and system-generated specifications (T = teacher-defined specification; G = system-generated specification). Values are reported per specification.	219
I.1	refinement was marked as successful when the refined solution and test cases executed without an error; this does not necessarily confirm their functional correctness. (T = teacher-defined specification; G = system-generated specification). Each specification was executed 30 times. Refinement success rate is computed over initially failing runs; final pass rate is computed over all runs.	226
J.1	Selected teacher questionnaire results across the three generated sequences.	231
J.2	Recurring themes in the teacher open responses.	232
J.3	Selected domain-expert questionnaire results across the three generated sequences.	232
J.4	Recurring themes in the domain-expert open responses and additional written feedback.	233

List of abbreviations and symbols

abbreviations

R G	Retrieval augmented system
DBR	Design based research
LLM	large language model

Symbols

Chapter 1

Educational context and problem setting

Educational context Biological and chemistry STEM-sciences (BCSW) is a study track in Flemish secondary education with a relatively small student population of approximately 1,300 students. Its limited scale provides little incentive for large educational publishers to develop extensive teaching materials specifically for this track.

This has practical consequences for teachers. They often cannot rely on a broad set of tailored resources. Instead, they combine material from different sources. They use online platforms, search the web for suitable exercises, adapt existing resources, or create material themselves.

This challenge has become more pronounced following the introduction of programming-related learning goals within the study track. Because these goals are relatively new, the availability of mature programming materials that align with both the curriculum and the BCSW context remains limited. Teachers in BCSW, therefore, face a double challenge. They work in a niche track with limited publisher support and for which suitable resources are still emerging.

Within this context, eTeacher is one example of existing digital support. The platform already offers a programming course that follows the learning goals set by the Flemish government. However, the exercise library is largely static. It was also not designed specifically for students in BCSW. Teachers using the platform have observed that some students question why programming is relevant to their studies. Some ask why they should learn it at all. Available exercises may therefore address the required programming concepts, without clearly connecting to the interests or perceived needs of this student group.

Teachers have expressed a need for programming materials that fit the BCSW context more closely. Conversations with teachers during this study also suggest that classrooms differ in important ways. Teachers not only vary in the type or difficulty of exercises they want, but also in their preferred pedagogical approach. Some favor project-based learning, while others prefer clearly structured learning paths. Others combine both approaches. In addition, some teachers want programming exercises

to align more closely with the world and domain context of their students.

Taken together, these observations point to a clear need for more customized programming context for BCSW. At the same time, BCSW is a relatively small study track. Any solution must therefore be viable and scalable. This creates a strong rationale for exploring more flexible approaches to content creation. Such approaches can respond to the specific needs of individual teachers and classrooms.

Problem Statement Existing programming exercises are often generic and insufficiently connected to the educational context of BCSW students. Although these exercises may address the required programming concepts, they may not clearly communicate how programming relates to the students' study domain.

Creating or adapting contextualized exercises manually requires considerable time and effort from teachers and is difficult to scale. Moreover, suitable exercises must do more than introduce a biological setting. They must remain aligned with the programming learning goals, match students' prior knowledge, and combine the programming task and biological context coherently.

This creates a need for a flexible generation approach that can produce interdisciplinary programming exercises while allowing educational requirements to remain controlled. Large language models offer the flexibility to generate varied content from structured instructions, but their outputs must also be evaluated for technical correctness, pedagogical suitability, contextual relevance, and practical usability.

Research objective The objective of this thesis is to investigate whether LLM-based generation can be used to create programming exercises that are both curriculum-aligned and meaningfully contextualized using biology concepts relevant to students in BCSW. In addition, this thesis aims to examine how the quality and usability of such exercises are evaluated by teachers and domain experts, and how students perceive their motivational value for learning programming.

Research questions

1. How can LLMs generate interdisciplinary programming exercises?
2. How do teachers and domain experts evaluate quality and usability?
 - Are the generated exercises pedagogically appropriate and contextually relevant?
 - How much do the teachers of BCSW need to adapt the generated exercise before use?
3. How did students experience the generated contextualized programming exercises, and what was their perceived impact on motivation?

Chapter 2

Literature Review

2.1 Theoretical frameworks for motivation and learning experience

Motivation is an important factor in learning, but it should not be treated as a single general construct. Students can engage with a task for different reasons. Some students may work because they find the task meaningful or interesting, while others may work mainly because of pressure, grades, or fear of failure. This distinction is important for this thesis because contextualized programming exercises aim to make programming tasks more meaningful for students.

This section first introduces Self-Determination Theory as a framework for understanding different types of motivation. It then discusses why the type of motivation matters in education. Finally, it introduces M-SRL to explain how students' motivation can interact with their experience of a specific learning task.

2.1.1 Self-Determination Theory

According to Ryan and Deci [7], Self-Determination Theory (SDT) offers a useful framework for understanding motivation in a more nuanced way than viewing it as a single measurable construct. Rather than treating motivation as something a person has more or less of, SDT distinguishes between different types of motivation and explains how the social environment and the self shape these. This distinction is relevant to this thesis because contextualized programming exercises are intended to help students recognize the relevance of programming within their study domain. SDT therefore supports the evaluation of whether students engage because they enjoy the exercises, recognize their value, experience external pressure, or perceive little reason to participate.

SDT proposes that motivation is influenced by the satisfaction of three basic psychological needs: autonomy, referring to a sense of choice and ownership; competence, referring to feeling capable of succeeding and making progress; and relatedness, referring to feeling supported and connected to others [7]. Satisfaction of these needs is generally associated with more self-determined forms of motivation

and greater well-being, whereas their frustration may contribute to controlled motivation or disengagement[7, 17]. Self-determined motivation refers to motivation that is experienced as coming more from the self, rather than from external or internal pressure.

Rather than treating motivation as a single construct, SDT places different forms of motivation on a continuum ranging from less self-determined to more self-determined forms of motivation, as shown in Figure 2.1 [7]. On the left side are forms of motivation that are experienced as less autonomous, with amotivation representing the most Nonself-determined form. On the right side are forms that are experienced as more autonomous, with intrinsic motivation representing the most self-determined form.

motivation refers to a lack of intention to act, for example because a person does not expect their effort to produce a meaningful outcome. External regulation describes participation driven by rewards, demands, or the avoidance of punishment. Introjected regulation is driven by internal pressure, such as guilt, shame, obligation, or the desire to protect one's self-worth. Although this pressure comes from within the individual, the activity is not yet fully experienced as personally chosen.

Identified regulation is more autonomous: the activity may not be inherently enjoyable, but the learner recognizes and accepts its value. With integrated regulation, this value has also become aligned with the individual's broader beliefs and sense of self. The distinction can be summarized as follows: identified regulation, "I do this because I think it is important", whereas integrated regulation means, "I do this because it fits with who I am." At the most self-determined end, intrinsic motivation refers to engaging in an activity because it is inherently interesting or enjoyable.[7]

External and introjected regulation are therefore considered forms of controlled motivation, whereas identified and integrated regulation, together with intrinsic motivation, are grouped under autonomous motivation.[7] This distinction provides the theoretical basis for the four motivational dimensions examined in the classroom study using the Situational Motivation Scale (SIMS): intrinsic motivation, identified regulation, external regulation, and amotivation[14].

2.1.2 Educational relevance of Self-Determination Theory

The distinction between these forms of motivation is important in education because students may demonstrate similar levels of effort while participating for fundamentally different reasons. Some students may engage because they find an activity interesting or recognize its value, whereas others may act primarily because of grades, pressure, guilt, or fear of failure. Observable effort alone therefore does not fully indicate the quality of a student's motivation.

This is illustrated by the large-scale meta-analysis [17], which examined 344 samples with more than 200,000 students from various countries. The study showed that the different motivational regulations distinguished by SDT have different relationships with educational outcomes[17, 13]. More generally, autonomous motivation is associated with beneficial outcomes such as engagement, persistence, achievement,

2.1. Theoretical frameworks for motivation and learning experience

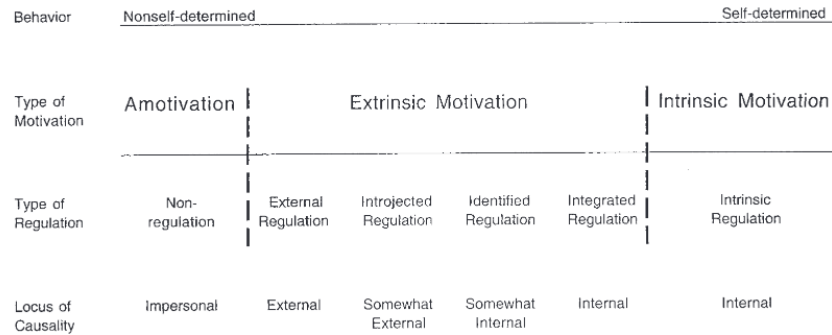


Figure 1. The self-determination continuum, showing the motivational, self-regulatory, and perceived locus of causality bases of behaviors that vary in the degree to which they are self-determined.

FIGURE 2.1: The self-determination continuum,[7]

and well-being, whereas controlled motivation and amotivation show less favorable or more variable relationships with these outcomes [17]. In that sense, motivation can be seen as a driving force behind learning, as it supports both performance and the ability to respond with challenges during the learning process [9].

More autonomous forms of motivation are generally associated with beneficial educational outcomes. Students who act because an activity is interesting or because they personally recognize its value tend to achieve better results [17][7], show greater persistence [17], and engage more deeply with the learning material[7]. They are also more likely to focus on understanding and mastering the material rather than simply completing tasks or outperforming others [13]. In addition, these forms of motivation are associated with more positive emotional experiences in school. It leads to greater enjoyment, interest, and a stronger sense of competence [7][13].

Controlled motivation can also lead students to invest effort. Students may participate because they want to obtain a reward, avoid punishment, meet expectations, or avoid feelings such as guilt and shame. However, this form of engagement is generally more fragile[7]. Research shows that these forms of motivation are often associated with anxiety, negative affect, and maladaptive coping.[13] Introjected regulation may still lead to persistence and observable effort, but this often comes at an emotional cost[13]. External regulation also does not reliably support academic performance or long-term persistence. Students can therefore appear motivated while experiencing the learning process as stressful or psychologically demanding[7][13].

motivation presents a different educational concern. It can occur when students do not expect their actions to produce a meaningful outcome or no longer believe that they are capable of succeeding. motivation is associated with limited engagement, low persistence, poorer achievement, and negative emotional experiences such as boredom and anxiety[17, 7]. Repeated experiences of confusion or

excessive difficulty may reinforce expectations of failure and reduce students' willingness to invest effort. Over time, students may stop believing that their actions can affect the outcome, which can contribute to learned helplessness. Preventing such experiences is therefore relevant not only to academic performance but also to students' confidence and psychological well-being[17][7].

Relevance to contextualized programming exercises. These distinctions are particularly relevant in secondary education, where grades, tests, and other external expectations can encourage participation without necessarily helping students experience a learning activity as meaningful. Educational design should therefore not focus exclusively on whether students complete an activity, but should also support their ability to understand its purpose, experience competence, and recognize its value.

Contextualization provides one possible way of making this value more visible. Connecting learning activities to students' interests, study domain, future goals, or possible real-world applications may support engagement and situational interest. In this thesis, connecting programming exercises to biology is therefore examined as a possible way of helping students recognize why programming may be relevant within their study domain. From an SDT perspective, this connection may particularly support identified regulation by making the value of programming more apparent.

However, contextualization does not automatically result in more autonomous motivation. Its effect depends on the chosen context and on how students experience the resulting exercise. In an unfamiliar context, an overly difficult programming task, or an unnecessarily long problem description may instead contribute to confusion or perceived difficulty. The educational opportunities and possible limitations of contextualization are discussed further in Section 2.2.3.

Within this thesis, SDT provides the main framework for examining the source and quality of students' motivation. It underlies the four motivational dimensions examined in the classroom study using the Situational Motivation Scale (SIMS): intrinsic motivation, identified regulation, external regulation, and amotivation. However, SDT provides less detail about how motivation interacts with cognition, emotions, perceived difficulty, and self-regulation while students work on a specific learning task. The following section therefore introduces the Metacognitive and Affective Model of Self-Regulated Learning to examine these task-level interactions.

2.1.3 Metacognitive and Affective Model of Self-Regulated Learning

While SDT describes the source and quality of students' motivation, the Metacognitive and Affective Model of Self-Regulated Learning (M²SRL) focuses on how motivation interacts with cognition, metacognition, self-regulation, and affect during a specific learning task[8]. In this thesis, M²SRL serves two main purposes. First, it functions as a preventive design lens by helping identify how features of an exercise may contribute to confusion, perceived difficulty, reduced confidence, or

disengagement. This provides guidance for limiting such risks during exercise design. Second, the model helps shape the student evaluation by drawing attention to task-level experiences such as perceived difficulty, confidence, interest, and students' perceptions of the context.

The M SRL framework

M SRL builds on the broader concept of self-regulated learning. Self-regulated learning refers to the processes through which learners monitor and adjust aspects of their cognition, motivation, behavior, and learning strategies in pursuit of their goals. This monitoring may involve conscious judgments, such as recognizing confusion or deciding to seek help, [8]. All learners engage in these processes to varying degrees. The differences lie in how effectively learners regulate their own learning [8].

M SRL places particular emphasis on how these processes interact. Furthermore, it highlights the role of meta-cognition and affect. Meta-cognition refers to learners' awareness and regulation of their own thinking. Affect refers to their emotional responses during learning. Together, these concepts help explain how learners monitor what is happening, how they feel about it, and how those reactions can lead to changes in strategy, effort, or behavior. M SRL structures this interaction through two related levels: the "person level" and the "task x person level" [8, 21].

Person level The "person level" consists of relatively stable characteristics that a learner brings into any learning situation. It includes traits such as general knowledge, self-concept, and metacognitive knowledge. Metacognitive knowledge refers to a learner's more stable beliefs about their own abilities, what a task demands, and useful strategies. These characteristics shape how learners set goals and plan their initial approach. [8, 21].

Task x Person level The "task x person" level refers to what happens while the learner is working on a specific task. At this level, learners form task-specific judgments and feelings, such as estimating how much time a task will take, or having a feeling of confusion or difficulty. These metacognitive experiences provide feedback about how the task is progressing and may lead learners to persist, change strategy, or seek help [8, 21].

An important aspect of these experiences is how learners interpret the difficulties they encounter. The same difficulty may be understood as a temporary challenge, as evidence that the current strategy is ineffective, or as an indication that the learner lacks ability. This interpretation is influenced by the more stable beliefs at the person level. As a result, learners may misattribute difficulty to themselves or to the subject, even when it is caused by the task formulation, the selected context, or an unfamiliar concept. Such interpretations can influence confidence, persistence, and future engagement.

Top-down and bottom-up regulation The two levels influence one another through top-down and bottom-up regulation. In **top-down regulation**, person-level knowledge, beliefs, and expectations shape how learners initially approach and interpret a task. For example, a student who believes that they are capable of programming may approach a difficult exercise with more confidence[8, 21]. In **bottom-up regulation**, experiences during the task lead learners to adjust their effort, strategy, goals, or help-seeking. These task-level experiences may be positive or negative. Repeated experiences can also gradually influence the more stable person level. Repeated success may strengthen a learner’s confidence or interest, while repeated difficulty may weaken these beliefs. Such changes generally develop across multiple experiences rather than through one isolated task[8, 21].

Relevance to the thesis M SRL is relevant to contextualized programming education because the effect of a context depends on the interaction between the exercises and the learner. Students bring prior knowledge, interests, beliefs, and expectations about both programming and the contextual domain. These person-level characteristics influence how the contextualized programming exercises will be experienced. M SRL helps explain how students interpret these experiences. For example, difficulty may be attributed to the programming concept, the biological context, the task formulation, or the learner’s own ability. This makes the framework useful both for anticipating possible problems during exercise design and for interpreting students’ experiences during the evaluation.

2.1.4 Takeaways

SDT and M SRL provide complementary perspectives on students’ engagement with contextualized programming exercises. SDT distinguishes between different sources and qualities of motivation. Therefore SDT is used to examine whether students engage because they enjoy programming, recognize its value, experience pressure, or see little reason to participate. M SRL complements this perspective by explaining how students’ more stable beliefs interact with their experiences during a specific exercise.

Together, the frameworks support both the design and evaluation of the exercises. They draw attention to the importance of making the value of programming visible while also considering how task features may contribute to confidence, interest, confusion, or perceived difficulty. Contextualization may support students in recognizing the relevance of programming within their study domain, but its effect depends on the selected context and on how students experience the resulting task. The pedagogical conditions and possible limitations of contextualization are discussed in the following section.

2.2 Pedagogical foundations for contextualized programming exercises

2.2.1 The pedagogical design challenge

Contextualized programming exercises aim to connect programming concepts to situations that students may perceive as meaningful. In this thesis, programming tasks are situated in a biological context. This can make the usefulness of programming more visible and may help students understand how programming can be applied outside a purely technical setting. However, adding a context also changes the demands of the exercise. Students must not only understand the programming concept but also interpret the biological information, identify how this information relates to the code, and understand the task instructions and expected output.

The design of contextualized programming exercises therefore requires a balance between relevance, clarity, and cognitive demands. The context should contribute to the purpose of the exercise without distracting from the intended programming goal.

At the same time, the programming task should match students' prior knowledge and provide sufficient support. This chapter first discusses the cognitive demands of contextualized learning tasks. It then examines contextualization and perceived authenticity, principles for introductory programming exercises, scaffolding, and feedback.

2.2.2 Cognitive demands of a contextualized learning task

Cognitive Load Theory explains that learning is limited by the capacity of working memory. New information must first be processed in working memory before it can be stored in long-term memory. However, working memory can only process a limited amount of new information at once[12]. Learning may therefore be hindered when a task requires students to process too many unfamiliar elements simultaneously[12].

The theory distinguishes between different sources of cognitive load. Intrinsic load refers to the complexity inherent to the learning task and depends on the concepts and relationships that students need to understand. Extraneous load refers to mental effort caused by the way the task is designed or presented. Unclear instructions, irrelevant information, poor structure, or unnecessary complexity may increase this load without supporting the intended learning goal. Cognitive overload occurs when the combined demands of the task exceed the learner's available working-memory capacity. In that situation, students may no longer have sufficient cognitive resources to focus on the intended learning content.[12]

This is especially relevant for contextualized programming exercises. These exercises combine programming concepts with a subject-specific context. This can make the task more meaningful, but it can also add extra information that students need to process[23]. If the context is unfamiliar, too complex, or poorly connected to the programming goal, it may increase extraneous load[23]. In that case, the context no

longer supports learning but competes with the programming concept for students' attention[23, 3].

The biological context should therefore not automatically be treated as either supportive or extraneous. Its effect depends on how it is presented, how closely it is connected to the programming goal, and how well it matches students' prior knowledge[23]. When the context helps students understand the purpose or structure of the problem, it can support the learning task[23]. When students must first interpret a complex or unfamiliar situation before they can reason about the programming concept, the context may instead compete for their attention[21].

Cognitive load is therefore an important design constraint in the development of contextualized programming exercises[23]. The context should complement the task without drawing too much attention away from the main learning goal[23]. The following section examines more closely how contextualization can support learning and why adding a context alone does not guarantee a successful learning experience.

2.2.3 Contextualization and perceived authenticity

Contextualization refers to connecting academic knowledge or skills to meaningful situations, practical applications, or domain-specific settings. Instead of presenting knowledge as abstract and detached from practice, it embeds learning within concrete settings that are relevant to students' lives, interests, or future goals[19, 28, 5]. This approach is rooted in cognitive science and situated cognition, which emphasize that knowledge is closely connected to the context in which it is learned and applied[19].

Within this discussion, a learner's **frame of reference** refers to the prior knowledge, experiences, interests, goals, and educational background through which the learner interprets a task. By connecting new knowledge to this frame of reference, contextualization may help students understand not only what they are learning, but also when and why that knowledge can be useful. This can reduce the risk of inert knowledge: knowledge that students can reproduce in a familiar educational setting but do not recognize or apply in other situations[19].

Different relationships between the subject matter and the context are possible. In contextualized instruction, one subject remains the primary learning goal, while another domain provides a setting in which that subject is learned or applied[28]. By contrast, in integrated instruction, the primary learning goal is the subject content itself, while the skill helps students engage with and understand that content[28].

n example to make the difference more concrete: In contextualized instruction, biology provides a meaningful domain for practicing programming concepts. In integrated instruction, programming is embedded as a tool to support the learning of biology content itself. This thesis focuses primarily on contextualized programming instruction. Programming remains the central learning goal, while biology provides the setting and subject matter through which programming concepts are introduced or practiced.

authenticity and perceived relevance

The presence of a context does not by itself guarantee that students will experience an exercise as meaningful. An important distinction must therefore be made between contextualization as a design choice and authenticity as a learner perception. A situation may be realistic from the perspective of a teacher or domain expert without being experienced as authentic or relevant by students[23]. Conversely, a simplified or fictional situation may still feel meaningful when students can connect it to their existing frame of reference. Personal relevance can therefore contribute to students' perceptions of authenticity, even when a task does not strictly reproduce a real-world situation [15].

Authenticity is consequently not an inherent property of an exercise. It depends on the relationship between the context, the task, and the learner. A context that is realistic but unfamiliar, uninteresting, or difficult to understand may not be experienced as authentic. By contrast, a simpler context may be perceived as meaningful when students understand its purpose and recognize a connection with their studies, interests, or future goals[23, 15].

This distinction connects contextualization to the motivational frameworks introduced in the previous section. A context that makes the purpose or usefulness of programming more visible may help students recognize the value of the activity. In terms of Self-Determination Theory, this may support identified regulation, even when they do not enjoy programming. Whether contextualization produces this motivational benefit, however, depends on how the context is selected and incorporated into the exercise. The following section therefore examines both the potential benefits of contextualization and the risks that may limit its effectiveness.

Potential benefits of contextualization

When contextualization is experienced as meaningful, it can improve several aspects of learning. It can increase the students' willingness to invest effort and persist. Furthermore, the students can become more engaged and motivated. When it is done well it provides a reason for learning an otherwise abstract concept[23]. In the context of this thesis, a biological setting may help students see how programming can support biological reasoning.

Contextualized instruction has also been associated with improved retention, transfer, and higher-order thinking because students are encouraged to connect knowledge to its possible use. Instead of practicing syntax in isolation, students are asked to interpret a situation, determine which information is relevant, and use programming to produce a meaningful result. This may support a deeper understanding of both the programming task and its possible applications[21, 23, 15, 11].

These outcomes should nevertheless be understood as potential benefits rather than automatic consequences of adding a context. Whether contextualization supports learning depends on how the context is selected, presented, and connected to the intended learning goal.

Risks and design conditions

Contextualization can also make a learning task more demanding. Students may need to understand unfamiliar terminology, identify which contextual information is relevant, and translate a domain-specific situation into a programming problem. When these additional demands are too high, they may contribute to extraneous cognitive load. This is especially relevant in interdisciplinary exercises, where difficulties may originate from either the programming content or the biological context[23, 21].

Students may, for example, interpret difficulty with a programming concept as evidence that they lack sufficient biological knowledge, even when the contextual information is not central to solving the task[21]. Similarly, an elaborate narrative may require students to process details that do not contribute to the programming goal. In such cases, the context competes with rather than supports the intended learning activity[4, 23].

Contextualization should therefore be treated as a design strategy with both opportunities and constraints. The context should connect to students' frame of reference, remain understandable with their existing domain knowledge, and have a functional relationship with the programming activity. At the same time, it should not introduce unnecessary information or obscure the intended learning goal.

Context can make a programming exercise more meaningful, but its inclusion alone does not guarantee a successful learning experience. The structure, difficulty, support, and clarity of the programming task remain equally important. The following section therefore examines the principles that guide the design of introductory programming exercises.

2.2.4 Design principles for programming exercises

Programming can place substantial demands on working memory because students must coordinate syntax, program logic, task requirements, and expected outputs. For novice programmers, processing these elements simultaneously can quickly lead to cognitive overload. The complexity and abstraction of introductory exercises should therefore increase gradually[12]. Programming also consists of several related but distinct skills. Students must learn how to read code, trace its behavior, understand syntax, recognize commonly used patterns, and apply those patterns to solve new problems [37]. Because these skills do not necessarily develop at the same pace, different exercise types, supporting materials, and teacher guidance need to work together in harmony. Students should therefore first learn to interpret and understand code before they are expected to produce complete solutions independently. Similarly, they should understand the role of an individual programming concept before applying it within a broader problem-solving task.

Progression from code understanding to problem solving

When a new programming concept is introduced, it is often best to begin with code reading and tracing. In these exercises, students inspect short examples, predict outputs, or explain how program state changes during execution. These activities

make the underlying reasoning visible and can reveal misconceptions before students begin writing their own code. Exercises can then gradually shift toward controlled code production, such as completing missing lines, modifying an existing fragment, or translating a short instruction into code[1, 12].

t this stage, tasks should remain focused. Constrained exercises reduce the number of decisions students must make and allow them to direct their attention toward the concept being introduced. Keeping the scope limited is especially important for beginners, who may otherwise can experience cognitive overload when they need to process too much at the same time[12, 37]. s students gain control over the concept and its syntax, exercises can gradually require more independent decisions and broader problem solving[37].

This progression should eventually move beyond practicing programming features in isolation. Many programming problems rely on recurring combinations of concepts that form reusable patterns[37]. For example, a while loop in combination with a stop condition and a counter. Exercises should help students understand how these elements work together rather than only how each element functions separately.

However, such patterns should also be introduced gradually[37, 12]. Students do not need to move directly from practicing basic while loops to the example mentioned above. n intermediate exercise may first combine a loop with a counter. later exercise can then introduce the if statement. In this way, the complete pattern emerges step by step, while each exercise builds on concepts that students have already encountered.

The progression from code comprehension to independent problem solving should not be interpreted as a fixed sequence that every exercise series must follow exactly. Instead, it provides a general design direction. Early exercises offer more structure and limit the number of choices, while later exercises increasingly require students to select and combine appropriate programming concepts themselves.

Clear and focused exercise design

Gradual progression alone does not make an exercise suitable for learning. Each exercise should also have a clear learning goal and remain focused on the concepts that students are expected to practice. n exercise may use an appropriate programming concept but still fail as a learning activity when it implicitly requires unfamiliar features or introduces avoidable complexity[12, 37]. The scope of the task should therefore match students' prior knowledge and the intended position of the exercise within the sequence[37].

The problem description plays a central role in this process. When instructions are vague, overly dense, or open to multiple interpretations, students may misunderstand the requirement and produce faulty solutions for reasons unrelated to the target concept [23, 34]. Problem statements should therefore use precise and unambiguous language. They should avoid unnecessary information and clearly describe what the program must do. Furthermore, they clearly state what input is given and what output is expected[12, 23].

Examples of inputs and expected outputs can make the required behavior more concrete and reduce ambiguity[12]. These examples should be consistent with the problem description and with any starter code provided to the student. The different components of the exercise should communicate the same requirements. Misalignment between the description, starter code, examples, and expected behavior can otherwise introduce confusion that is unrelated to the learning goal[12].

Introductory programming exercises should therefore develop students' skills gradually, move from comprehension toward independent production, and help students recognize how individual concepts form reusable programming patterns. At the same time, each task should remain focused on a clear learning goal and communicate its expected behavior through precise instructions and representative examples. Even when these conditions are met, beginners may still require support while applying unfamiliar concepts. The following section examines how scaffolding and feedback can provide this support while gradually encouraging greater independence.

Scaffolding in programming exercises

Scaffolding is an important design principle in introductory programming education. They lower the entry barrier and help students focus on the intended reasoning process [1]. Scaffolding provides temporary support that helps students work through a task. The goal is not to remove the difficulty of the exercise, but to make the task manageable enough for students to focus on the intended learning goal[12, 1].

Scaffolding can take different forms in programming exercises. In this thesis, the most relevant forms are starter templates, completion problems, step-by-step task structures, explanatory comments, and hints[1, 12]. These scaffolds can be applied both in the problem statement and in the code that is provided to the student. They can support students by providing code structure, clarifying the problem-solving process, or pointing them toward a relevant programming pattern.

The educational value of scaffolding lies in the way it directs students' attention. A well-designed scaffold reduces unnecessary difficulty while preserving the core reasoning step of the exercise[1]. For example, a starter template can already provide the function structure, while still requiring students to decide which condition, loop, or calculation is needed. In this way, the scaffold does not solve the exercise for the student, but helps the student focus on the programming concept that the exercise is meant to practice. In contextualized exercises, scaffolding can also help students connect the context, the task description, and the code without being overwhelmed by all elements at once.

Scaffolding should change across an exercise sequence. Early exercises may contain stronger support, such as partially completed code, explicit steps, or guiding comments. Later exercises should gradually remove part of that support, so that students need to make more decisions independently. This fading of support helps students move from guided practice toward more independent problem solving[1]. For example, an early exercise may provide the loop structure and ask students to complete the condition, while a later exercise may only provide comments that hint

on what steps to take. In this way, scaffolding supports progression rather than keeping students dependent on the same level of guidance.

However, more scaffolding is not always better. When too many forms of support are combined in one exercise, the support itself can become an additional source of complexity[38]. A task may include a worked example, a detailed starter template, several hints, and reflection prompts. Each of these elements can be useful separately, but together they may require students to process too much information before they can begin solving the programming problem. Excessive scaffolding can lead to higher risk of cognitive overload[38] or passive engagement, where students follow the provided steps mechanically without understanding the structure of the solution[18]. In some cases, it may encourage students to guess and check until the answer is accepted[18].

For this thesis, scaffolding is therefore treated as a design variable that must be controlled across an exercise sequence. Generated exercises should not simply include as much support as possible. The amount and type of support should depend on the intended programming concept, the assumed prior knowledge of the students, and the position of the exercise in the sequence. Good scaffolding reduces avoidable difficulty, but still leaves enough room for students to reason, make decisions, and develop their own problem-solving strategies.

Feedback in programming exercises

Feedback should be considered part of the design of programming exercises[37]. In programming education, delayed feedback can slow down learning because students may repeat the same mistake without understanding why their solution fails. Immediate feedback can reduce this problem by helping students detect errors earlier and revise their solution while the task is still active[?]. This is especially relevant for beginning programmers, who may struggle to identify whether a problem is caused by syntax, program logic, or a misunderstanding of the task[25]. Feedback can therefore support revision and help students maintain a sense of progress[34, ?].

However, feedback should do more than indicate that an answer is wrong. A correctness signal alone gives limited support when students do not know what to inspect next. Useful feedback should be clear and actionable: it should help students locate the issue and understand the likely cause[39]. Otherwise, feedback risks becoming a signal to guess again rather than a support for learning[?]. In automatically supported programming exercises, test cases are one way to provide this kind of immediate feedback[34]. They can check selected behaviors and provide messages that point students toward likely mistakes[34, ?]. At the same time, this feedback remains limited because tests only cover anticipated cases.

Pedagogical requirements in this thesis

The literature reviewed in this section shows that the quality of a contextualized programming exercise cannot be reduced to whether its code executes correctly. Technical correctness is a necessary baseline, but the exercise should also function

as a coherent learning task. Its problem description, code, examples, scaffolding, feedback, and expected behavior should communicate the same requirements and remain aligned with the intended learning goal.

Pedagogical quality further depends on whether the task matches students' prior knowledge, introduces new demands gradually, manages cognitive load, and provides appropriate scaffolding and actionable feedback. Contextual quality concerns whether the biological content is accurate and understandable for the intended students, functionally connected to the programming activity, and experienced by students as authentic or meaningful. These aspects interact. A context may, for example, be scientifically accurate but still be pedagogically unsuitable when it is too complex or distracts from the programming goal. Similarly, a context may be correctly integrated into the exercise without necessarily being perceived by students as authentic or relevant.

A generated exercise should therefore be evaluated as a complete learning task. Its components must be technically reliable and coherent with one another while supporting the intended pedagogical purpose and providing a context that is accurate, functionally relevant, and meaningful to students.

These requirements create a particular challenge for automatic generation. A generation system must not only produce the required exercise components, but also preserve their alignment with the learning goal, assumed prior knowledge, selected context, and intended level of support. The following section examines how existing approaches use prompting, structured input, grounding, decomposition, validation, and feedback to support these qualities and where additional control remains necessary.

2.3 Automatic generation of educational programming exercises

The previous section established that contextualized programming exercises must be technically reliable, internally coherent, pedagogically appropriate, and meaningfully contextualized. The challenge for automatic generation is how system design can help produce and safeguard these qualities when LLM outputs remain probabilistic. This section reviews existing approaches to automatic exercise generation and considers how their design choices can help control the generated material.

Automatic exercises and question generation (AQG) is not a new idea; the field has existed for quite some time. Earlier systems used rules, feature-based methods, templates, and neural networks[20, 31]. These approaches show that educational content generation has long aimed to reduce manual authoring work, but they also illustrate the difficulty of combining control, flexibility, and educational alignment[20].

Most recently, Large language models (LLM) transformed the field. They can generate richer educational content from natural-language and structured instructions[31, 24]. This flexibility is useful for contextualized programming exercises because the design dimensions discussed above can vary across classrooms, topics, and exercise

sequences[33, 27]. The next subsections, therefore, discuss the mechanisms used to guide, ground, validate, and refine LLM-based generation.

2.3.1 LLM-based generation: fine-tuning and prompting

LLM-based generation systems can be adapted to educational tasks in different ways. A useful distinction is between approaches that adapt the model itself and approaches that keep the model fixed but guide it through the input. Fine-tuning belongs to the first group. It further trains an existing model on task-specific examples, so that the model becomes more specialized for a particular kind of output. Prompt-based generation takes the second route. Here, the model is not retrained. Instead, the generation process is steered through instructions, constraints, examples, and contextual information.

Fine-tuning can be valuable when the target task is clearly defined and when enough representative training examples are available[31]. For example, if a system repeatedly generates the same type of exercise in a stable format, fine-tuning can help the model learn the expected structure and style. However, this approach becomes less suitable when the desired use of the system is still being explored.

A fine-tuned model can support variation, but this variation must be sufficiently represented in the training data. For contextualized programming exercises, this is difficult because the generator may need to respond to different learning goals, classroom situations, subject contexts, and forms of teacher control. Moreover, the exact relation between teachers and the generation tool is still part of the design problem[27]. It is not yet fully clear which controls teachers need, how much freedom they should have, or how such a tool should fit into their workflow[27].

Prompt-based generation therefore offers a more suitable starting point for this type of exploratory design. It makes it possible to translate pedagogical intentions, domain expertise, and practical requirements into structured instructions for the model. The prompt can specify what the exercise should focus on, what it should avoid, which context it should use, and how the output should be structured. In this way, the prompt becomes a control mechanism between the educational design and the generated material.

For contextualized programming exercises, this control is important. The generator should not only produce a technically correct programming task. It should also keep the programming goal central, use the biological context in a meaningful way, match the assumed prior knowledge of students, and fit within a broader learning sequence. Prompt-based generation makes it possible to adjust these requirements during development without first defining a complete training dataset for every expected use case. At the same time, it shifts the main design challenge to the prompt and the surrounding generation process. The quality of the output depends on how clearly the task is specified, how relevant the provided context is, and how the generated result is checked and refined.

The following subsection, therefore, discusses the mechanisms used to guide and control prompt-based generation. These mechanisms include instruction design, examples, knowledge grounding, decomposition, feedback, and tool use.

2.3.2 Controlling prompt-based generation

first way to steer prompt-based generation is through the role or profile assigned to the model. The model may, for example, be instructed to respond as a teacher, teaching assistant, domain expert, or reviewer[22]. Such role descriptions can influence the perspective from which the model approaches the task[29, 35]. Studies show that the effect of a profile prompting is not always consistent across models and tasks[22]. However, it remains valuable to steer the generation toward a specific goal[22, 29]. This is especially important in educational settings, where the model should respond from a pedagogical perspective rather than produce general output. Its value lies less in guaranteeing correctness and more in guiding the model toward an appropriate perspective for the task.

role description is not sufficient on its own. The task instructions must also make the goal and constraints explicit[29]. Good instructions define what the model should produce. It contains three elements: intent, demand, and domain[29]. Intent expresses the main purpose of the task[29]. Demand adds more detailed requirements and constraints[29]. Domain specifies the context in which the task should be interpreted[29]. Together, these elements provide a useful structure for writing effective instructions[29]. They encourage the user to reflect on the task, the requirements for the output, and the contextual information that must be made explicit. This matters because assumptions that may seem obvious to humans are not necessarily made by the model. In that sense, instruction design can be seen as a way of translating human expertise into explicit and organized instructions that the model can follow.

Examples can provide an additional form of guidance. In few-shot prompting, the prompt includes one or more examples that show the desired output structure[29]. In Exgen, few-shot prompting improved the generation of more usable programming exercises compared to zero-shot prompting, which more often leads to trivial or duplicated outputs [33]. At the same time, research identifies instruction design as a strong alternative to few-shot prompting. Clear instructions can impose stricter constraints and offer more direct control. In addition, it avoids higher token cost and sensitivity to example selection [22]. Still, few-shot prompting often remains more effective when the model must imitate a specific and difficult-to-describe output structure.

Instructions and examples are therefore complementary. Instructions define the purpose and constraints of the task, while examples demonstrate the expected structure or style. For educational generation, this combination is useful because the output often needs to satisfy both formal and pedagogical requirements. A programming exercise should contain the right components, but these components should also fit together as a coherent learning task.

Grounding generated educational content

A well-structured prompt does not guarantee that the model uses the most relevant or accurate knowledge. LLMs can produce fluent output that contains inaccurate

information, unsupported claims, or terminology that does not match the intended learning context[22]. In educational applications, this is problematic because generated material should align with the concepts, definitions, examples, and level used in the course.

Knowledge grounding addresses this limitation by adding relevant external information to the generation process. Retrieval Augmented Generation (RAG) is one common approach. In such systems, relevant fragments from course material, domain documents, or other external knowledge sources are retrieved and added to the prompt before generation.[22] The model can then base its output on information that is more closely aligned with the educational context.

RAG pipeline typically follows three broad steps. First, external information is indexed in a searchable form[29]. For example, in a structured database or a vector database. Second, when a query is submitted, the system retrieves the most relevant information. Third, this information is added to the prompt. Then the response is generated using that prompt. Although the exact implementation of RAG may differ. These three stages remain largely the same. In practice, each stage can be optimized separately. For example, changing how the data is stored can have an impact on how relevant the information is when it is retrieved.

In educational exercise generation, grounding can support alignment with course terminology, reduce generic output, and limit the risk of inaccurate or inappropriate content. This becomes especially important when exercises include a subject-specific context. In that case, the context should not function as a loose theme or keyword. It should provide information that is accurate, understandable, and meaningfully connected to the task students are expected to complete.

2.3.3 Decomposition in prompt design and generation workflows

Another way to guide prompt-based generation is decomposition. Decomposition tries to mirror human problem-solving by breaking down a complex problem into smaller steps. This can happen within a single prompt or across multiple prompts. In both cases, the purpose is to make the generation process more explicit.[29].

Within a single prompt, decomposition can be used as an instruction-writing strategy. The prompt may ask the model to first identify the learning goal, then consider the relevant constraints, and only afterwards generate the final output. In this approach the intermediate reasoning steps and final answer are generated together.[29] Chain-of-thought prompting is one example of this broader idea, because it encourages the model to work through intermediate reasoning steps before producing an answer[29]. In educational generation, this can help make implicit design considerations more explicit.

Decomposition can also be implemented at the workflow level. In that case, a complex generation task is divided across several connected prompts or agents, where the output of one step becomes the input for the next[32]. One step may generate an initial exercise, another may review the output, and another may refine it. This is often referred to as prompt chaining or multi-agent generation[32]. Multi-

agent systems such as PyTaskSyn illustrate this approach by assigning different roles to different agents in the generation process[27].

The distinction between these forms of decomposition is important. Decomposition within a prompt structures the reasoning of one generation step. Decomposition across prompts structures the generation workflow itself. Both forms can be useful for educational generation because educational outputs often need to satisfy several requirements at the same time. generated programming exercise may need to include a clear problem statement, suitable code, a correct solution, relevant examples, test cases, and feedback. Decomposition can help separate these concerns and reduce the risk that one part of the task receives too little attention.

Implication for educational exercises generation

The literature shows that LLMs can generate educational material, including programming exercises, but also that generation quality depends on how the task is specified and structured. Roles, instructions, examples, grounding, and decomposition provide different ways to guide the model toward more useful output. They help define what should be generated, which constraints should be respected, which knowledge should be used, and how complex generation tasks can be made more manageable.

For contextualized programming exercises, these mechanisms are especially relevant. The generated output must combine programming content with a subject-specific context while keeping the programming goal central. It must also remain understandable for the target students and produce components that can be checked or reused. Prompt-based generation provides the flexibility needed for this kind of variation, but it also creates a need for additional safeguards. The following subsection therefore discusses how validation, feedback, and tool use can further support the reliability of generated exercises.

2.3.4 Validation, feedback, and tool use

Like mentioned before LLMs are probabilistic systems, their outputs can vary and may contain mistakes. The mechanisms discussed above are ways to reduce this uncertainty within the prompt itself. Still they do not remove the need to check the generated output. Even when a prompt contains clear instructions, examples, contextual information, and decomposition steps, the generated exercise could still be incomplete, inconsistent, or incorrect. In programming exercises, these problems can appear at different levels. The generated code may not run, the test cases may not match the model solution, required output fields may be missing, or the structure may not be suitable for automatic processing. At the same time, an exercise can be technically valid while still being pedagogically weak, unclear, or poorly aligned with the intended learning goal.

first way to increase reliability is to use tools for tasks that do not need to be handled by the LLM itself[22]. In this context, tools are external components that perform a specific operation outside the model. Their role is not limited to

checking the final output. They are especially useful when a task can be handled more reliably by a specialized or deterministic system than by natural-language judgment[22]. For example, instead of asking an LLM whether a program works, the system can compile and run the program. Similarly, generated test cases can be executed against a model solution. Furthermore, deterministic checks can be used to verify whether the output has the required format or contains all required fields. In these cases, the LLM remains responsible for generation and interpretation, while the tools provides a more reliable check for specific aspects of the output. This makes it possible to detect problems before the exercise is shown to a teacher or used in a platform.

However, technical validation is not sufficient for educational exercise generation.

program may execute correctly while the task description remains vague. test suite may pass while the feedback messages are not helpful for students. biological context may sound plausible while using imprecise terminology or introducing a misconception. These aspects require interpretation rather than execution alone. For that reason, feedback agents can be used as an additional LLM-based quality-control mechanism[29]. Feedback can be generated by the same model that produced the exercise, but this has limitations. When a model evaluates its own output, it may overlook mistakes or reinforce assumptions made during generation[29]. separate feedback agent can therefore provide a more independent review.[29] feedback agent is an LLM-based component that evaluates generated output from a specific perspective, such as a teacher, student, or domain expert[22]. In this way, feedback agents can help check whether the generated exercise satisfies not only technical requirements, but also the pedagogical and contextual requirements discussed in the previous sections.

Validation tools and feedback agents therefore address different aspects of the same quality problem. Tools are useful when part of the output can be checked through execution, deterministic rules, or another specialized system. Feedback agents are useful when the quality of the exercise depends on interpretation rather than direct verification. Together, they make the generation process more controlled. Feedback and tools are especially useful in multi-prompt or multi-agent pipelines. When each step depends on the previous one, a small error early in the chain can affect all later outputs. Therefore, it is better to catch such problems as early as possible. This enables stopping the pipeline, preventing avoidable token costs.

Alternatively, this can trigger a refinement loop, where the problematic output is revised before the workflow continues.

Validation and feedback can improve the reliability of generated educational material, but they do not remove the need for human judgment. Teachers remain important for judging whether generated material fits their students, lesson goals, and teaching approach. However, validation and feedback can improve the draft before it reaches that stage. Instead of asking teachers to identify every technical, contextual, or pedagogical issue themselves, the system can already filter out part of these problems during generation. This is important because the aim is not only to produce exercises automatically, but to produce material that can serve as a usable starting point for classroom adaptation.

2.3.5 Teacher-oriented design and usability of generated materials

The practical usability of an educational generation system depends not only on whether it can produce educational material, but also on whether teachers can readily incorporate the resulting output into their practice. Generated material must fit the intended curriculum, students, and classroom situation, while remaining sufficiently transparent and adaptable for teacher review. Usability therefore concerns both the suitability of the output and the work required to make it suitable for a particular educational setting.

Research involving K–12 teachers indicates that generative- I tools can support lesson planning and material creation by allowing content to be produced and adapted more quickly[10]. t the same time, their practical usefulness can decline when generated outputs do not match the local curriculum, students’ level, or the cultural and educational context. The value of generation consequently depends not only on its efficiency, but also on whether the resulting material fits the setting in which it will be used[10].

Selwyn et al. (2025)[30] show more directly how these limitations affect teachers’ work. Materials produced using general-purpose generative- I tools frequently require checking, correcting, reorganizing, or rewriting before they can be used. This work is not caused only by factual inaccuracies. Teachers must also determine whether an output reflects the local curriculum, their pedagogical intentions, the needs of particular students, and the circumstances of the lesson. Because some of this knowledge is contextual and unstated, repeated prompt refinement does not necessarily produce classroom-ready material. Generated outputs therefore remain drafts requiring teacher review.[30]

Quality and usability are therefore closely related, but not identical. Problems in the technical, pedagogical, or contextual quality of an exercise generally increase the revision work required from teachers. Nevertheless, material of acceptable general quality may still require adaptation to fit a particular classroom or lesson. These findings suggest that educational generation systems can support teachers by giving them greater control over the intended output and by reducing avoidable revision work, while leaving the final judgment about classroom use with the teacher.

2.3.6 Existing generation engines

Existing LLM-based generation systems illustrate how some quality and usability concerns can be addressed through system design. ExGen uses few-shot prompting to guide the generation of problem statements, reference solutions, and test cases.

difficulty-assessment agent subsequently acts as a filter: exercises that do not pass this assessment are not retained for use[33]. PyTaskSyn extends this idea through a multi-agent pipeline, where role-specific agents simulate expert, tutor, and student perspectives. The expert agent generates a contextualized programming task, while tutor and student agents evaluate its correctness, contextual relevance, and comprehensibility [27]. These systems show that LLM-based exercise generation can be combined with validation and review steps, rather than relying only on a

single direct generation step.

Although it does not focus specifically on programming exercises, Intelli provides another relevant example of teacher-controlled educational generation. Instead of requiring instructors to construct complete prompts manually, the system allows them to provide source material and configure pedagogical parameters such as the instructional goal, question format, and difficulty. These selections are subsequently translated into prompts by the system. Intelli therefore illustrates how structured input can give instructors control over relevant characteristics of the generated material while reducing their reliance on manual prompt engineering. However, the system was evaluated within a university course, and its applicability to secondary programming education and interdisciplinary exercise generation remains to be established.[36]

These systems illustrate different ways of controlling educational generation. These systems illustrate different approaches to controlling educational generation. ExGen uses examples to guide generation and a difficulty-assessment filter to exclude unsuitable exercises[33], PyTaskSyn distributes generation and evaluation across agents representing different educational perspectives[27], and Intelli translates structured pedagogical choices into prompts[36]. This thesis builds on these approaches within a more specific educational setting. Exercises are not treated as independent outputs, but as parts of an educational design process in which teachers need control over what is generated and how it fits within a learning progression.

Existing programming-exercise generators commonly steer contextualization through keywords, topics, or themes. This supports variation, but provides less control over whether the context is grounded in structured domain knowledge and aligned with the surrounding curriculum. Intelli offers more explicit control over pedagogical characteristics, but does not address contextualized programming exercises whose different components and position within a sequence must remain aligned.

The approach examined in this thesis therefore investigates how structured teacher input, domain grounding, generation and evaluation components, and technical validation can be combined to support both the quality of the generated exercises and their practical usability for teachers.

Chapter 3

Materials and methods

3.1 Study design and scope

This thesis follows an exploratory design-based research approach to develop and evaluate an LLM-based system for generating contextualized programming exercises. The study is organized into four connected phases, as summarized in Figure 3.1. These phases cover the exploration of the design context, the iterative development of the exercise-generation layer, the evaluation and abstraction of the resulting design knowledge, and its subsequent application and human-centered evaluation.

The first phase identifies teacher needs, classroom constraints, and technical platform requirements through the exploratory study and the literature review. These findings are translated into system requirements that guide the subsequent development process. In the second phase, the exercise-generation layer is developed through six iterative cycles. Each cycle combines implementation with internal evaluation and refinement. Two external review rounds provide additional feedback at selected points in this process. The first review informs further system-level development, while the second identifies remaining limitations after the introduction of dedicated test generation.

In the third phase, the resulting exercise-generation layer is evaluated system-

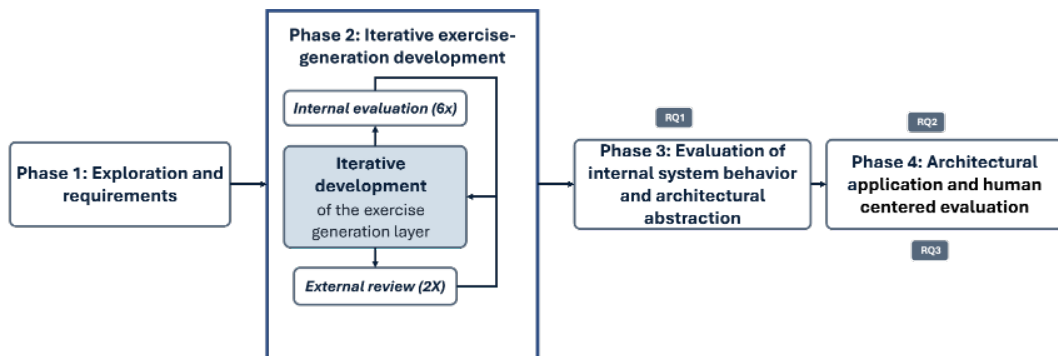


FIGURE 3.1: Overview of the four study phases, the iterative development of the exercise-generation layer, and their relationship to the research questions.

atically by examining its internal behavior, including output consistency and the operation of its feedback, refinement, and execution mechanisms. The findings from this evaluation are combined with the design knowledge accumulated throughout the six development cycles. Recurring functions and relationships are then abstracted into a general generation architecture. This architecture represents the main system-level contribution of the thesis and is used to address RQ1

The fourth phase applies the general architecture to sequence generation and evaluates the generated materials from the perspectives of their intended users. The architecture is used to develop a sequence-generation layer that produces exercise blueprints. These blueprints subsequently serve as structured input for the exercise-generation layer, which converts them into complete exercise sequences. Separately, for the classroom study addressing RQ3, students from the educational master’s program created structured exercise blueprints that served as direct input to the exercise-generation layer. The sequence-generation layer was therefore not used for this study. The detailed organization of the classroom study is explained in Section 3.7.1

The different phases therefore serve complementary purposes. The iterative development and internal system evaluation examine how an exercise-generation system can be constructed and controlled. The classroom study and teacher and domain-expert evaluation examine the generated material from student, teacher, and domain perspectives. Given the exploratory scope of the study, these evaluations are used primarily to identify strengths, limitations, and opportunities for further refinement rather than to establish broad claims about classroom effectiveness.

3.1.1 Design based approach

Design-based research connects theoretical principles with their practical implementation in authentic educational settings. It supports the iterative development and examination of an intervention while accounting for the requirements and constraints of the context in which it is intended to be used[16, 26]. It is characterized by iterative cycles of design, evaluation, reflection, and refinement. These cycles serve not only to improve the intervention but also to identify why particular design components are needed and how they contribute[16, 26]. These activities are translated into the four study-specific phases presented in Figure 3.1

Phase 1 corresponds to the preliminary exploration and establishes the literature-informed and stakeholder-informed requirements for the system. Phase 2 represents the design and development activity, conducted through six iterative cycles. Phase 3 performs the retrospective analysis by consolidating the findings from the development process and internal system evaluation into a general generation architecture. Phase 4 subsequently applies this design knowledge to sequence generation and extends the evaluation to students, teachers, and domain experts.

In accordance with DBR, the iterative cycles are intended both to improve the exercise-generation layer and to produce design knowledge. Major additions and modifications respond to needs or limitations identified through the literature, exploratory study, internal evaluation, or external review. Evaluation and reflection

therefore occur throughout development, rather than only after the implementation has been completed. The retrospective analysis in Phase 3 consolidates the knowledge accumulated across these cycles and abstracts it into more general architectural components and relationships.

3.2 Exploratory phase: design context

3.2.1 Teacher needs and classroom constraints analysis

Teacher needs and classroom constraints are explored using an exploratory survey and semi-structured interviews. The survey is completed by six teachers. The interviews involve four participants: two informatics teachers and two professionals who support teachers through pedagogical guidance and professional-development activities. The interviews provide more detailed insight into classroom experiences and implementation constraints, while the survey collects input from a broader group in a structured manner.

The interviews provide more detailed insights into teachers' experiences and constraints, while the survey collects broader input in a more structured way. Both instruments focus on teachers' background, the organization of informatics learning goals, existing teaching materials, contextualization practices, student motivation, and implementation constraints.

Because of the limited number of participants, the findings are used to identify recurring needs and concerns rather than to estimate how commonly these occur among all BCSW teachers. These findings inform the input and output requirements, contextualization approach, and evaluation criteria of the generation system. The survey and interview guide are included in [appendix 3.2.1](#).

3.2.2 eTeacher platform analysis

Because the main intervention takes place in eTeacher, the platform is analyzed before the generation engine is designed in detail. This analysis does not evaluate eTeacher as a learning platform. Instead, it identifies the practical and technical constraints that the generated exercises need to follow.

The analysis focuses on three aspects. First, existing eTeacher modules are reviewed to understand how programming concepts are introduced and ordered. This helps identify where the intervention can fit within the existing learning trajectory. Second, the supported activity types are documented. Although eTeacher supports different types of activities, this thesis focuses on code-based programming exercises. Other activity types are therefore considered part of the broader platform context, but are not included in the generated output. This limits the scope of the thesis, which supports an easier evaluation. Third, the technical structure of code-based exercises is analyzed, including the required exercise components, Y ML structure, and test case format.

Together, these observations define the platform-related constraints for the generation engine. Since the exercises are intended for use in eTeacher, the system not

only needs to generate pedagogically meaningful exercises. It also needs to produce the components required by code-based eTeacher exercises and structure them in a way that can later be converted to the platform format.

3.2.3 Collaboration with educational master students

In addition to the above-described analysis, the design process includes collaboration with students from the educational master's program. These students design a small curriculum intervention with contextualized programming exercises. This intervention aims to explore whether connecting programming concepts to biological content can support the motivation of BCSW students.

The collaboration works as follows. The educational master students define the pedagogical design of the exercise sequence. This includes decisions about the learning goal, the order of the exercises, the intended progression, and the way the biological context supports the programming concept. The generation engine uses this information as input and generates exercises that fit these requirements. In this way, the system does not generate exercises from a broad prompt only, but from explicit pedagogical decisions that describe the intended role of each exercise in the sequence.

The educational master students also review the generated exercises during development. Their feedback is used to check whether the exercises follow the intended pedagogical structure and whether the biological context is used in a meaningful way. This collaboration helps define the input structure of the exercise generation layer and supports the refinement of the generated exercises during development.

3.3 From design context to system requirements

The exploratory phase and literature review were translated into a set of requirements for the exercise generation layer. This layer is responsible for generating complete programming exercises from structured input. The requirements define the information required before generation, the artifacts that must be produced, the functionality the generation layer must provide, and the criteria that generated exercises must satisfy. Together, these requirements guide the design of the generation architecture described in the following sections. The sequence generation layer is introduced later as an extension of this architecture and therefore has additional sequence-specific requirements.

3.3.1 Input requirements

The input requirements define the pedagogical and technical information that must be provided before generation starts. These requirements enable teachers to specify the desired characteristics of the generated exercise or exercise sequence. Table 3.1 summarizes the required inputs and their design rationale.

Input (provided by teacher)	
Input parameter	Design rationale
Main programming topic	Defines the primary programming concept the exercise should teach or practice.
Additional programming concepts	Specifies other concepts the teacher would preferably like to include.
Allowed programming concepts	Defines which supporting concepts may be used in the exercise.
Context topic + guidance	Contextualizes the programming exercise with a meaningful and grounded context (e.g., Biology).
Prior knowledge + guidance	Enables the generated exercise to be built upon previously acquired knowledge.
Exercise type	Allows teachers to choose between different pedagogical activities.
Output (generated by the system)	
Output artifact	Design rationale
Problem statement	Presents the programming task within a meaningful biological context.
Template code	Supports scaffolding by providing an appropriate starting point for students.
Model solution	Supports teacher review and provides a reference implementation for automated validation.
Test cases	Enable automatic assessment and immediate formative feedback.
Feedback messages	Help students interpret test outcomes and guide further learning.
Metadata	Preserves pedagogical information required for platform integration and sequence planning.
Platform representation	Converts generated exercises into a format compatible with the eTeacher platform.

TABLE 3.1: Inputs to the generation engine and the resulting output artifacts.

3.3.2 Output requirements

The output requirements define the artifacts produced by the generation engine. Together, these outputs support students during the learning process, provide teachers with the information required to review generated exercises, and enable integration with the eTeacher platform. Table 3.1 summarizes the generated artifacts and their design rationale.

3.3.3 Functional requirements

Beyond defining the required inputs and outputs, the generation engine should satisfy several functional requirements. These requirements capture the capabilities

Functional requirement	Design rationale
Context-aware generation	Produces exercises that integrate programming with meaningful biological contexts.
Pedagogical alignment	Ensures generated exercises match the specified learning objectives, difficulty, and scaffolding level.
Sequence awareness	Supports coherent learning progressions across multiple exercises.
Automated quality assurance	Detects and refines pedagogical, biological, and technical issues before deployment.
Platform compatibility	Enables direct integration into the eTeacher platform and existing teaching workflow.
Extensibility	Supports future adaptations, additional domains, and new exercise formats through a modular architecture.

TABLE 3.2: Functional requirements of the generation engine and their design rationale.

required to generate pedagogically meaningful, technically correct, and practically usable programming exercises. Table 3.2 summarizes the functional requirements together with the design rationale that motivated their inclusion.

3.3.4 Evaluation dimensions and criteria for generated output

The output format describes what the generation engine aims to produce, whereas the evaluation criteria describe the aspects used to assess the generated exercises.

Although related, they serve different purposes: the output format guides the system design, while the evaluation criteria provide a structured framework for assessing the quality and usability of its output.

The literature review identifies technical quality, pedagogical quality, contextual quality, and usability as four broad evaluation dimensions. Based on the literature and the exploratory findings, these dimensions are further specified through criteria relevant to the teaching context and system examined in this thesis. Table 3.3 groups these criteria under their corresponding dimensions. Together, the dimensions and criteria provide a consistent framework for system development and evaluation.

Although presented separately, the quality dimensions and usability are related. Technical, pedagogical, or contextual limitations can reduce usability by increasing the review and adaptation required before classroom use. However, material of sufficient quality may still require adaptation because it does not fit a particular curriculum, lesson structure, or classroom context. Usability therefore depends partly on the quality of the generated material and partly on its fit with the teacher’s intended use.

TABLE 3.3: Evaluation framework used to assess the quality and teacher usability of generated exercises and sequences.

Evaluation dimension	Evaluation criterion	Focus of the evaluation
Technical and structural quality	Technical quality	Correctness, executability, and internal consistency of the problem statement, template code, model solution, and test cases.
	Output format compliance	Presence of the required components and compliance with the prescribed output structure.
Pedagogical quality	Programming alignment	Alignment with the intended programming concept, prior knowledge, and defined scope.
	Problem statement quality	Clarity, completeness, conciseness, and understandability for the target students.
	Exercise type fit	Correspondence with the intended exercise type and level of scaffolding.
	Sequence coherence	Fit with the exercise's position in the sequence and progression from earlier exercises.
	Student guidance	Extent to which hints, examples, and test feedback support understanding and help students interpret mistakes.
Contextual quality	Contextual quality	Scientific correctness and relevance of the biological context, and its meaningful connection to the programming task.
Teacher usability	Curriculum and classroom fit	Fit with the curriculum, students' level and prior knowledge, available teaching time, and the teacher's lesson organization and instructional approach.
	daptation effort	Amount of reviewing, correcting, reorganizing, supplementing, or rewriting required before the generated material can be used in the classroom.

3.4 Prompt development methodology

3.4.1 Prompt design process

The system contains several LLM-based components for generation, evaluation, and refinement. Although these components perform different tasks, their prompts are developed using the same general process, shown in Figure 3.2. The figure illustrates the four-step logic used for creating a complete prompt-based agent and for developing or modifying an individual prompt section. In both cases, the process begins by defining the goal of the component. Based on this goal, the required output is specified. These requirements are then translated into a prompt structure. Finally, the generated output is evaluated against the intended goal and requirements, after which the prompt is refined where necessary.

This approach mainly builds on the profile and instruction dimension and the GPEI framework. The profile and instruction dimension guides the structure of the prompt itself. It helps define the profile of the agent, the intent of the task, and the instructions that guide the output. The GPEI framework provides the broader iterative design logic: goal defining, prompt construction, evaluation and refinement. Other prompt engineering frameworks and guidelines are also reflected in the prompt sections. However, only selected elements from these approaches are applied, rather than the full frameworks themselves. For this reason, they are not treated as separate design frameworks in this thesis.

As previously stated, the same development process was used at two levels. At the component level, it was applied to complete prompt-based components, such as generation, evaluation, and refinement agents. At the section level, it was used to create or revise individual prompt sections that support a specific part of the component’s task. In that case, no separate profile or intent is defined, because these are inherited from the larger prompt in which it is used. Instead, the section is designed to support a specific part of the agent’s task. For example, a section can define output constraints or contextualization rules.

This makes prompt design more systematic. Each prompt or prompt section is linked to a specific goal and expected output. When evaluation reveals a recurring limitation, the corresponding instruction or section is adjusted without redesigning

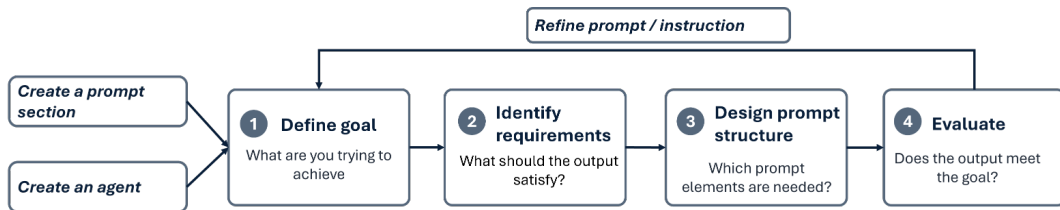


FIGURE 3.2: Prompt design process used for creating agents and prompt sections. The process starts from a defined goal, translates this goal into output requirements and prompt structure, and uses output evaluation to refine the corresponding prompt or instruction when needed.

the complete prompt. This also makes it possible to relate prompt changes directly to observed output problems during the iterative development of the system.

3.5 System architecture

3.5.1 architecture development approach

The system architecture was developed through the design-based research process introduced in Section 3.1.1. The exercise-generation layer served as the starting point. Rather than defining the complete architecture in advance, the system was refined through successive cycles of generation, evaluation, and redesign.

The requirements identified in Section 3.3 guided this development, but they were not translated into a predefined set of components at the start. Instead, they were implemented progressively as the development process revealed how they could be translated into concrete architectural components. Each iteration therefore introduced additional components where greater control, structure, or validation was required.

The development process includes qualitative evaluation throughout the different design iterations. Each internal iteration follows the same general cycle. First, the generated output is evaluated against the requirement and evaluation criteria defined in Section 3.3. Recurring shortcomings were analyzed and used to revise the architecture in the subsequent iteration. Generation logs were consulted when necessary to determine where observed problems originated within the pipeline. The process therefore combines the implementation of new components with repeated assessment of whether the generated exercises satisfy evaluation criteria and requirements.

Two external review rounds supplemented this internal formative evaluation. For both reviews, the material consisted of the first 21 exercises generated from structured input defined by students from the educational master project group. The exercises were exported to Markdown and PDF and reviewed from both biology-education and programming-education perspectives. The reviewers provided written and oral feedback.

The first review took place after the introduction of modular prompt construction and contextualized generation. Its findings informed the subsequent feedback-based refinement iteration. The second review followed the introduction of dedicated test generation. It identified remaining limitations and informed adjustments to individual exercises, but did not result in another architectural iteration within the scope of this study.

The following sections first describe the iterative development of the exercise-generation layer. The recurring structure that emerged from these iterations is then generalized into a reusable generation architecture. Finally, this architecture is adapted to create the sequence-generation layer, which produces structured exercises requirements that are inputs for the exercises generation layer. These structured exercise requirements will also be called blueprints from now on.

3.5.2 Iterative development of the exercise-generation layer

The exercise-generation layer transforms structured requirements into complete programming exercises. It was developed through six successive iterations. Each iteration introduced additional structure, validation, or platform support. This section summarizes the architectural changes; the observations motivating them are discussed in Section 4.1

Baseline generation

The first version used a single prompt to generate a beginner-level programming exercise containing a problem statement, solution, and test cases. The output was produced as unstructured text and served as the initial baseline.

Structured generation and execution

The second version introduced a fixed JSON output schema so that exercise components could be processed separately. CodeRunner executed the model solution and generated tests, while execution failures triggered an initial refinement step.

Modular and contextualized generation

The static prompt was replaced by a prompt builder that combines reusable sections from a prompt library. Retrieval components added relevant biological information and selected earlier exercises to support grounded contextualization and continuity between exercises.

Feedback-based refinement

The external feedback from previous iteration and the evaluation criteria from Section 3.3 were translated into teacher- and student-oriented feedback perspectives. These perspectives evaluated aspects that could not be checked through execution alone. The teacher perspective focuses on the scientific and pedagogical quality of the exercise, while the student perspective focuses on clarity, usability, and the expected experience of the learner. When revision was required, their feedback was passed to an apply-feedback component that generated a refined exercise.

Dedicated test generation

Test-case generation was separated from the main exercise-generation step. A dedicated component generated tests from the finalized problem description and model solution, after which the tests were executed for technical validation.

eTeacher translation

The final iteration added a platform-specific test-generation component and a deterministic mapper. Together, these convert the finalized platform-independent JSON exercise into the YML structure required by eTeacher.

Together, these iterations transformed the exercise-generation layer from a single prompt into a modular pipeline containing generation, contextual grounding, qualitative feedback, technical validation, refinement, and platform-specific formatting.

3.5.3 Final exercise- generation engine

The iterations described above resulted in the final exercise-generation layer, which transforms structured requirements into complete programming exercises. Relevant biological information and, when applicable, selected earlier exercises are first added to the generation context. The main generation component produces the exercise content, after which a dedicated component generates the internal test suite. These outputs are combined into a structured draft.

The draft is then evaluated through feedback agents, code execution, and deterministic validation. When revision is required, the resulting feedback is applied before the exercise is finalized. The platform-independent JSON output is stored in the exercise library and can later be retrieved when referenced by another exercise. For deployment in eTeacher, a separate platform-specific test-generation step and deterministic mapper produce the required Y ML representation.

cross these components, prompts are constructed through the modular prompt builder introduced during the iterative development. The builder combines reusable prompt sections and inserts task-specific values from the structured input, allowing the same prompt structure to be adapted for different generation and evaluation tasks.

The final exercise-generation layer was used to produce the material evaluated in the subsequent studies. Its components also revealed a broader functional structure, which was formalized as the general generation architecture presented in Section 4.4 and later reused to construct the sequence-generation layer.

3.5.4 dapting the general architecture for sequence generation

The sequence-generation layer applies the general functional structure identified during the development of the exercise-generation layer Section 4.4. Instead of generating complete exercises directly, it generates the exercise blueprint which is required by the exercise-generation layer. Each generated blueprint follows the input structure defined in Section 3.3. It specifies the intended programming focus, exercise type, biological context, prior knowledge, and other constraints for one exercise in the sequence.

Before blueprint generation begins, a sequence-initialization step enriches the initial sequence input. It determines the relevant programming subtopics, prior programming knowledge, contextual direction, and progression guidance for the sequence. Its output is added to the context library, from which relevant information is selected and inserted into the prompt during blueprint generation.

The individual blueprints are then generated sequentially using two generation components. The first component generates the programming and pedagogical elements of the blueprint. Based on these elements, a second component generates

the corresponding contextual elements. The sequence-initialization output provides shared guidance for both stages, supporting alignment between the programming goals and the biological context. Similar to the separation of test-case generation in the exercise-generation layer, this decomposition separates concerns between the programming and pedagogical structure of an exercise and its contextualization.

During generation, the system uses a sliding window of previously generated blueprints. This allows later blueprints to build on recent exercises without providing the complete generation history. The same window constrains prior-knowledge references, since a blueprint can only refer to earlier exercises that are available within the current context.

The evaluation components evaluate whether a blueprint is suitable for downstream exercise generation. The prior knowledge evaluator checks whether the selected previous exercises are a meaningful reference. The context evaluator checks whether the biological context supports the programming goal. The programming and pedagogy evaluator checks whether the difficulty progression remains gradual. Deterministic evaluation components verify the required blueprint structure, context references, and prior-exercise identifiers. After refinement, the finalized blueprints are stored and passed to the exercise-generation layer, where they serve as generation input.

3.6 Internal system evaluation (phase 3)

Phase 3 evaluates the internal behavior of the final exercise-generation layer through three complementary analyses. The consistency analysis examines how the generated exercise components vary when the same teacher-defined or system-generated input is processed repeatedly. The feedback analysis examines when the feedback agents request refinement, which components are modified, and how the initial draft differs from the finalized exercise. The execution analysis records the outcomes of the execution step that is already part of every generation run. It examines whether the generated tests pass with the corresponding model solutions, which errors occur, and how the system responds internally to these outcomes. Together, these analyses assess the stability, refinement behavior, and technical robustness of the generation pipeline. They do not directly establish the pedagogical quality or classroom usability of the generated material, which are examined through the human studies in Phase 4.

3.6.1 Evaluation setup

The evaluation uses the final exercise-generation workflow described in Section 3.5.3. Two input types are considered: teacher-defined blueprints and blueprints generated by the system. For each condition, five input blueprints are used. Each blueprint is processed by the exercises generation layer 30 times using the "gemini-3-flash-preview" model, resulting in 150 generated exercises per condition and 300 generated exercises in total. The model configuration and generation settings are kept constant

across runs so that observed differences mainly reflect variation in the model output rather than changes in the experimental setup.

For every run, the initial and finalized exercise components, feedback-agent verdicts, applied refinements, execution outcomes, and execution errors are recorded. When code-related components are refined and executed again, both the initial and subsequent execution outcomes are stored. This data provides the shared basis for the consistency, feedback-agent, and execution analyses.

3.6.2 Component-comparison measures

Different comparison measures are used for the textual, code-based, and structured components of the generated exercises.

Problem-description The consistency of the problem descriptions is evaluated at two levels. First, textual similarity is measured using BLEU and Python’s SequenceMatcher. BLEU measures overlap in words and short word sequences and therefore reflects similarity in local phrasing and terminology. SequenceMatcher compares the complete character sequences and identifies longer matching blocks, providing a broader indication of overall textual resemblance and structural reuse. Using both metrics reduces reliance on a single representation of textual similarity and makes it possible to distinguish between repeated local phrasing and wider similarity in the structure of the descriptions.

Second, semantic similarity is measured using sentence embeddings. Each problem description is converted into a vector representation using the all-MiniLM-L6-v2 model, after which cosine similarity is calculated between pairs of vectors. This measure captures whether two descriptions express a similar meaning even when they use different wording. The all-MiniLM-L6-v2 model was selected because it is a lightweight model and is commonly used for cosine-similarity comparisons. Together, the textual and embedding-based measures distinguish between consistency in phrasing and consistency in meaning.

Code measures The template code, model solution, and generated test cases are compared using CodeBLEU. The metric combines n-gram overlap, weighted n-gram overlap, and syntactic similarity based on the abstract syntax tree. In this evaluation, the metric uses weights of 0.33 for n-gram matching, 0.33 for weighted n-gram matching, 0.34 for syntactic matching. CodeBLEU is selected because it captures both textual and structural similarities in generated Python code more effectively than a purely text-based metric.

Metadata measures Structured concept metadata are compared using Jaccard similarity. This metric expresses the overlap between two sets of concepts as the size of their intersection divided by the size of their union. A score of one indicates identical concept sets, whereas a score of zero indicates that the sets contain no shared concepts.

3.6.3 Consistency analysis

For each input blueprint, the 30 finalized exercises are compared pairwise, resulting in 435 unique comparisons. Each pairwise comparison produces a similarity score for the corresponding exercise component. These scores are then aggregated per blueprint and per input condition to provide an overview of the consistency of teacher-defined and system-generated inputs. Higher scores indicate that repeated generations produce more similar outputs, whereas lower scores indicate greater variation.

Spearman rank correlations are calculated between the individual pairwise similarity scores of the different components. These correlations examine whether variation in one component is associated with variation in another component.

3.6.4 Feedback and test case refinement behavior

The feedback and execution data are summarized per exercise blueprint type. The analysis examines how often the feedback agents request a revision, how often execution fails, and how often refinement resolves these flagged issues or execution failures. Draft and final versions are compared using the same component-specific similarity measures described above, and are used to determine which exercise components are changed most frequently during refinement.

To examine whether the refinements are necessary and meaningful, random subsets are drawn from the exercises that undergo refinement. Up to five exercises per blueprint are selected separately for each blueprint type. This is done for both feedback- and execution-based refinement. This results in 2 subsets containing feedback refinement behavior and 2 subsets containing execution-based refinement behavior. In total 82 exercises were looked at. Their initial and final versions are reviewed to determine what prompted the refinement, whether the identified issue was addressed, and whether the change introduced another limitation.

3.7 Human-centered evaluation (phase 4)

Phase 4 extends the evaluation from the internal behavior of the generation system to the generated material as assessed by human participants. This phase examines the material that the system can produce through two complementary studies. The classroom study addresses RQ3 by examining how students experience the gener-

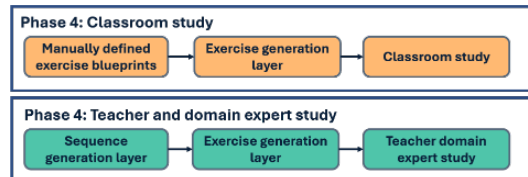


FIGURE 3.3: Generation workflows used for the classroom study and the teacher and domain-expert evaluation.

ated exercises and their biological contexts, including their perceived motivational impact. The teacher and domain-expert study addresses RQ2 by evaluating the quality and educational usability of generated exercise sequences from the perspectives of teachers and domain experts. Together, the studies identify strengths, limitations, and opportunities for further refinement.

s summarized in Figure 3.3, the studies use different generation workflows. For the classroom study, the educational master project group manually designed the exercise blueprints, which were subsequently processed by the exercise-generation layer. For the teacher and domain-expert study, the complete two-layer process was used: the sequence-generation layer first generated the exercise blueprints, after which the exercise-generation layer generated the exercises. Participants evaluated only the resulting exercise sequences, not the intermediate blueprints or individual generation steps. Their evaluations therefore concern the quality and usability of the final output rather than the performance of either layer in isolation.

Both studies are exploratory and involve relatively small, context-specific samples. The findings are therefore interpreted as indications of patterns, strengths, limitations, and opportunities for system refinement rather than as causal evidence or broadly generalizable conclusions about classroom effectiveness.

3.7.1 Classroom study

The classroom study includes 43 BCSW students from three schools. Students are randomly assigned to a control group of 22 students and a contextualized group of 21 students. The control group completes decontextualized programming exercises, whereas the contextualized group completes biologically contextualized versions.

Intervention material and procedure

The intervention material consists of a sequence of 23 programming exercises. The exercises were generated using the exercises blueprints designed by the project group of the educational master program. After the generation, the educational master's students made small manual adjustments where needed. These adjustments mainly focus on improving classroom usability. For example, examples that were too similar to the actual exercise content are replaced. In some cases, additional scaffolding is added to basic problem-based exercises because the transition from highly scaffolded exercises to more open problem-based exercises would otherwise be too large for students.

To compare contextualized and non-contextualized material, a de-contextualized version of the same sequence is created by hand. In this version, the biological context is removed while the programming focus remains as comparable as possible.

The study is conducted through three separate one-hour classroom sessions. During each session, students completed their assigned version of the programming exercises on the eTeacher platform. The pre-intervention questionnaire was administered before students began the exercises, and the post-intervention questionnaire was administered after the intervention.

Data collection

The pre-intervention survey measures their situational motivation toward "*Informat-icawetenschappen*" using the Situational Motivation Scale (SIMS). It is a validated questionnaire for measuring situational motivation.[14] Students answer each item using a seven-point scale. The questions cover intrinsic motivation, identified regulation, external regulation, and amotivation. The post-intervention survey repeats the SIMS items to support comparison with the pre-intervention measurement.

Students in the contextualized group answer additional questions about the biological context using a five-point scale. These questions are informed by M-SRL, cognitive-load theory, and perceived authenticity. These questions examine students' perceived experiences across the categories defined in Table 3.4. The exact questions are included in appendix B.

Data analysis

The analysis consists of two parts. First, the comparative motivation analysis conducted in the educational master study is summarized. Second, the students in the contextualized group are examined in more detail to investigate how they experience the generated exercises and how these experiences relate to their motivational orientation.

The educational master study compares the contextualized and control groups using Mann-Whitney U tests and examines pre- to post-intervention changes within each group using Wilcoxon signed-rank tests. This analysis uses an exploratory significance level of $\alpha=.10$. The present thesis reports only the main findings and refers to the educational master's paper[6] for the complete statistical procedure and results. These findings are interpreted as possible patterns in students' perceived motivation, rather than as evidence of a causal motivational effect.

The additional analysis focuses on the 21 students in the contextualized group. Their responses to the five-point questionnaire items are analyzed descriptively using counts and percentages. To connect the analysis to the theoretical framework presented in Sections 2.1 and 2.2, the items are organized into four exercise-experience categories, see Table 3.4. These categories are used only to organize the analysis and are not treated as validated scales. Each questionnaire item is therefore analyzed separately.

Motivation and exercise experience The pre- and post-intervention SIMS scores are summarized descriptively for the contextualized group. Spearman's rank correlations are used for two exploratory analyses. First, correlations are calculated between the individual exercise-experience questionnaire items to examine how different aspects of students' perceptions of the exercises relate to each other.

Second, selected exercise-experience items are related to identified regulation using Spearman's rank correlation. Identified regulation is selected because it reflects whether students recognize the value or importance of an activity and therefore most directly corresponds to the intended motivational role of the contextualization. In

TABLE 3.4: Exercise-experience categories used to organize the post-intervention questionnaire items for the contextualized group.

Exercise-experience category	Questionnaire focus
Perceived interest and preference	Whether the context makes the exercises more interesting and whether students prefer exercises without context
Perceived usefulness and purpose	Whether the context clarifies the usefulness of programming and helps explain what the program should do
Feasibility and cognitive burden	Whether the programming level is feasible and whether students experience unnecessary complexity, overload, or frustration
Understandability of the context	Whether the biological context is easy to understand

addition to the correlations with post-intervention motivation, a secondary analysis examines whether perceived usefulness is associated with the change in identified regulation. This change is calculated by subtracting the pre-intervention score from the post-intervention score. For each correlation, Spearman’s ρ change, the number of included students, and the exact p-value are reported.

3.7.2 Final teacher and domain expert evaluation

Materials and Participants

The evaluation material consists of three sequences focusing on variables, conditionals, and while loops. Each sequence contains five exercises and introduces its respective programming concept. Participants receive the learning goal, assumed prior knowledge, and in-scope and out-of-scope concepts for each sequence, allowing them to assess the material in relation to its intended purpose.

Five participants contribute to the evaluation. Two teachers complete the teacher questionnaire: one is preparing to begin working as a teacher, while the other has four years of experience teaching informatics in the fifth and sixth years of secondary education. The teacher with 4 years experiences also participates in a semi-structured follow-up interview.

Two domain experts complete the domain-expert questionnaire. They both have twelve years of experience in computer science and programming and limited to moderate experience evaluating beginner programming exercises. A third domain expert, who is completing a PhD in computer science, reviews the sequences and provides feedback by email without completing the questionnaire. Questionnaire results are therefore based on two teachers and two domain experts, while the interview and email feedback provide supplementary qualitative evidence.

Data collection and analysis

The same sequences are assessed through separate teacher and domain-expert questionnaires. Both questionnaires contain five-point agreement items, an ordinal item concerning the extent of required adaptation, and open-response questions. Although they overlap in the aspects assessed, they examine these aspects from different perspectives. The teacher questionnaire emphasizes student learning and classroom use, whereas the domain-expert questionnaire places greater emphasis on programming-related quality. Some exercise components are therefore assessed from both perspectives. For example, teachers consider whether a template provides appropriate support for students, whereas domain experts consider whether it is technically correct.

The instruments assess selected evaluation criteria within the evaluation dimensions presented in Table 3.3. The questionnaire items and interview prompts to these dimensions and criteria is provided in appendix C.

The follow-up interview elaborates on the teacher’s questionnaire responses and examines how the sequences could be incorporated into teacher-led instruction. The interview prompts serve as a flexible guide rather than being asked verbatim or in a fixed order.

Questionnaire responses are summarized descriptively using counts and response distributions. Adaptation ratings are retained in their original ordinal categories. Open responses and interview notes are organized according to the applicable evaluation dimensions and criteria. The email feedback is treated as supplementary qualitative evidence and discussed alongside the relevant findings in the Results and Discussion.

Chapter 4

Results and discussion

4.1 Findings from the exploratory study

The exploratory phase examined the educational and technical context in which the programming exercises would be used. It combined a survey completed by six teachers, three semi-structured interviews, and an analysis of the eTeacher platform and its technical documentation. Because of the limited number of participants, the findings are interpreted descriptively and are used to identify relevant design needs rather than to represent all BCSW schools. Detailed survey and interview results are provided in [appendix D](#), while the technical eTeacher structure is documented in [appendix E](#).

4.1.1 Educational context and material needs

Figure [4.1](#) summarizes several key findings from the survey. The survey showed some variation in how informatics was organized in the participating schools. Five of the six respondents reported that the informatics learning goals were taught in a separate subject, while one respondent indicated that they were integrated into STEM or science. Platform use also differed: three respondents used eTeacher, one used another digital platform, and two did not use a specific platform. The interviews provided further context for these differences. Participants indicated that the implementation of informatics depended on factors such as the school, educational network, study track, teacher expertise, available infrastructure, and number of teaching hours.

Rather than pointing toward one fixed instructional setting, these findings indicate that generated material should remain adaptable to different classroom organizations and teaching practices. This is consistent with the literature on teacher-oriented educational generation, which emphasizes that generated material needs to fit local curricula, student groups, and classroom situations rather than function as universally applicable output[\[10\]](#). For the generation system, this supports exposing relevant educational choices to the teacher instead of assuming a fixed lesson structure.

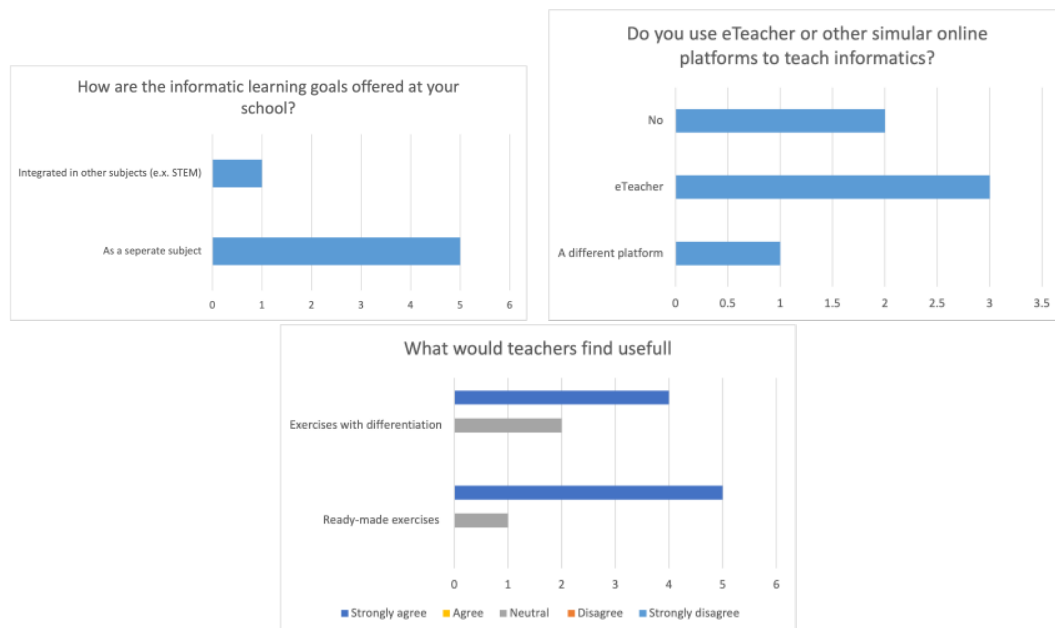


FIGURE 4.1

The survey also showed interest in additional teaching support. Five of the six respondents gave the highest rating to ready-made exercises as a form of support for contextualized informatics lessons. Four gave the highest rating to exercises with differentiation, while the remaining two selected the middle response category. The interviews supported this finding. Participants described the preparation of suitable teaching material as time-consuming, particularly when exercises had to be developed from scratch. They expressed a need for material that could serve as a usable starting point while remaining adaptable to the level of the students, and the teacher's preferred approach. Recurring needs included clear problem descriptions, model solutions, testing possibilities, differentiation, and support for teachers with different levels of programming experience.

These findings suggest that the purpose of the generator should not be to remove teachers from the design process, but to reduce the amount of material they need to create from scratch while preserving control over important characteristics of the output. This directly motivated the use of structured input parameters for programming focus, prior knowledge, exercise type, allowed concepts, and contextual guidance, as defined in Section 3.3

4.1.2 Programming instruction and contextualization

The interview participants described introductory programming instruction as a gradual process involving different types of learning activities. Small and clearly defined exercises were considered particularly suitable when students first encountered a programming concept. One described progression began with an explanation or simple example, followed by jointly constructing a program and only afterwards mov-

ing toward independent work. Participants also mentioned activities such as, tracing code, creating a flowchart or another form of preparatory analysis before coding, completing partially provided code, and modifying an existing program rather than starting from an empty file. Larger projects and synthesis exercises were considered more appropriate after students had acquired sufficient foundational knowledge.

The survey showed a similar preference for gradual progression. For assignments that build step by step towards a larger whole, five of the six respondents selected category 4 or 5 across each of the four programming areas included in the questionnaire. The interviews additionally highlighted difficulties with combining multiple concepts, translating an abstract problem into programming steps, debugging, and differences in students' prior knowledge and pace.

These observations align closely with the literature reviewed in 2.2, which emphasizes gradual progression, alignment with prior knowledge, focused exercise design, and scaffolding that is reduced as students become more independent. Rather than treating these principles as characteristics that the LLM should infer implicitly, the findings therefore support representing them explicitly in the generation process. The system should know which concepts students already encountered, which concepts are allowed in the current exercise, what type of exercise is intended, and how much support the exercise should provide.

Biological and scientific contextualization was viewed as potentially valuable because it could make the application of programming more visible. Participant referred to examples such as DNA sequences, disease models, ecological models, and scientific visualizations. The open survey responses identified several conditions for successful contextualization. Respondents emphasized clear learning objectives, familiar biological content, a meaningful relationship between biology and programming, feasibility within the available lesson time, and sufficient examples and support.

The interviews similarly stressed that the biological or scientific information needed to be correct and should not obscure the programming objective. Unfamiliar terminology, unnecessary information, or substantial additional domain knowledge could make an exercise more difficult. This corresponds with the literature on contextualization and cognitive load discussed in Section 2.2: context can make the purpose of a task more visible, but it can also become an additional source of difficulty when it is unfamiliar, excessive, or poorly connected to the intended programming goal.

For the generation system, contextualization should therefore not be implemented as simply adding a biological theme to a programming exercise. The context needs to be grounded in appropriate domain knowledge, constrained by what students are expected to understand, and functionally connected to the programming task. These findings motivated the inclusion of explicit contextual guidance in the generation input, together with later mechanisms for contextual grounding and qualitative evaluation.

4.1.3 Technical constraints identified in eTeacher

The eTeacher analysis showed that course material follows a predefined hierarchy in which modules contain chapters, chapters refer to section files, and sections contain ordered learning activities. These activities can include explanatory text, examples, questions, thinking activities, and programming exercises. This allows conceptual explanation, guided activities, and programming practice to be combined within one learning sequence. However, the generation system developed in this thesis is limited to code-based programming activities

These constraints meant that the generation system could not treat an exercise as unrestricted generated text. The individual exercise components needed to remain separately accessible and structurally consistent so that they could later be translated into the representation expected by eTeacher. This directly supported the move toward a structured intermediate representation and, later in the development process, a separate platform-specific mapping step. Detailed Y ML fields, test structures, and an example representation are provided in [appendix E](#).

Together, the exploratory findings and literature review therefore translated the educational context into the input, output, functional, and evaluation requirements defined in [Section 3.3](#). The next phase examines how these requirements were progressively implemented through the iterative development of the exercise-generation layer.

4.2 Iterative development of the generation architecture

TABLE 4.1: Overview of the architectural iterations and their resulting capabilities.

Iteration	Main observed limitation	rchitectural change	Resulting capability
Baseline	Unstructured and inconsistent output	Static prompt and structured format	More reliable output structure
Modular generation	Limited control and contextual grounding	Modular prompt and retrieval	Reusable instructions and grounded context
Feedback refinement	Qualitative issues remained	Teacher and student feedback agents	Pedagogical and contextual review
Dedicated tests	Tests conflicted with solutions or the execution environment	Separate test-generation component	Focused technical validation
Platform integration	Internal output differed from e-Teachers	Mapping and platform-specific tests	Deployable Y ML output

4.2.1 Purpose of the iterative analysis

As described in Sections 3.5 and 3.6.1, the exercise-generation layer was developed through repeated cycles of generation, formative evaluation, and redesign. The generated exercises were assessed using the requirements and evaluation criteria defined in Section 3.3. When an issue in the final output could not be explained directly, the generation logs were consulted to identify where it arose in the pipeline.

This section reports the main observations from these iterations and explains how they led to changes in the generation architecture. Table 4.1 summarizes the main limitations identified during each iteration, the corresponding architectural changes, and the resulting capabilities. The subsections that follow provide more details per iteration.

4.2.2 Baseline and static prompt

Baseline The baseline iteration examined whether a language model could generate the main components of a beginner-level programming exercise from limited instructions. The generated output included a title, task description, solution, test cases, and a list of programming concepts. This showed that a single prompt could produce a coherent and human-readable exercise.

However, the baseline output also shows why additional structure is needed. The exercise is generated as free-form Markdown, making it difficult to extract and process its components automatically. More importantly, the model independently introduced programming concepts beyond the intended learning goal. In the baseline example, the exercise is about loops, but it also uses input handling, conditional statements, type conversion, exception handling, randomness, and Boolean logic. These additions are not necessarily wrong, but it was not what was intended for a beginner exercise.

The baseline established that direct LLM generation was feasible, while showing the need for more structured and controlled generation.
appendix F.1.1

Static The second iteration introduced structured JSON output and code execution. The individual exercise components could therefore be extracted and processed separately. The generated solution and test cases could also be executed, and an execution failure could trigger refinement. When the generated solution fails one or more test cases, the model is prompted to revise the exercise and solution based on the execution feedback.

Contribution to the architecture This represented an important technical improvement: generation, execution, and correction could be combined within one pipeline. It introduced structured output and execution-based validation as basic components of the generation pipeline.

Remaining limitation Successful execution only showed that the solution and tests were technically consistent with each other. It did not show that the exercise is pedagogically well scoped or that the solution only uses programming concepts that fit the intended learning goal. The structured prompt also became increasingly difficult to maintain as more requirements were introduced. Supporting allowed concepts, exercise types, contextual information, prior knowledge, and sequence position within one static prompt would make later adaptation difficult. This motivated the transition to modular prompt construction. [appendix F.1.2](#)

4.2.3 Modular prompt construction and contextualized generation

The third iteration shows a generation process that is more controllable than the static prompt version. The generated exercises follow the requested structure more consistently and are easier to adapt to different input choices. Instead of mainly asking the model to generate an exercise about a broad topic, the system can now give more precise constraints about the programming concept, the allowed concepts, the exercise type, the biological context, and the relation to earlier exercises.

The inspected outputs showed that these constraints helped contain the conceptual scope of the exercises. They also enabled the biological context to become a functional part of the programming task. For example, list indexing fits naturally with retrieving a DNA base from a list, and strings fit naturally with DNA sequences as ordered character sequences. In these cases, the biological context gives meaning to the data, while the programming goal remains central. [appendix F.2.1](#)

Continuity between exercises A second important result is that a generated exercise could refer to previously introduced concepts or reuse a structure that students had already encountered. This made it possible to generate connected exercises rather than only separate tasks presented in a chosen order. This makes the generated material more than a collection of separate exercises placed after each other. For example, look at the exercise sequence in [appendix F.2.1](#) here it can clearly be seen that exercises make references to what was seen before.

Contribution to the architecture Prior-knowledge input gives the generation engine a way to refer back to earlier tasks, reuse previously introduced logic, and position the new exercise more clearly within the learning path. Modular prompt construction allowed programming constraints, exercise type, and other requirements to be incorporated through reusable prompt sections rather than one increasingly complex static prompt. The retrieval mechanism made it possible to select relevant external information and integrate it into the generation context when needed, including biological domain knowledge and information from earlier exercises. Together, these mechanisms made the generation process more adaptable and controllable while allowing relevant information to be incorporated only when needed.

Remaining limitation found through the first external review Despite these improvements, the first external review identified recurring quality problems.

Several comments concerned biological precision and terminology. For example, some exercises described DNA as being constructed directly from nucleobases rather than nucleotides containing nucleobases. Other formulations, such as “four important bases,” could incorrectly imply that additional DNA bases exist.

Reviewers also recommended more precise terminology. For example, the feedback suggests using “nucleobases” instead of “letters” when the biological meaning is important. It also suggests replacing “Drosophila” with “Drosophila melanogaster” or “D. melanogaster” when referring to the model organism. The review also identified issues outside the biological context. One example contains an incorrect expected output in a sequence-comparison exercise. These findings showed that improved control over generation input did not provide sufficient qualitative quality control after generation. This motivated the introduction of feedback-based refinement. The complete notes from the first external review are provided in appendix F.6.1.

4.2.4 Feedback-based refinement

The fourth iteration no longer treats a generated exercise as an output that only needs to be technically valid. Instead, it is treated as a draft that is reviewed from different perspectives before it is finalized. The external feedback and the evaluation criteria from Section 3.3 were translated into teacher- and student-oriented feedback perspectives. These perspectives evaluated aspects that could not be checked through execution alone.

The feedback-agent logs show how the implemented agents can identify small issues and convert them into targeted revision suggestions. The logs also show that refinement is not always needed, since several generated exercises pass the feedback step without further changes.

One example where refinement was needed is shown in appendix F.3.1. In this case, the feedback agent assigns “MAJOR REVISION” because the exercise states that DNA is translated to proteins and uses DNA triplets with T as codons. The feedback points out that codons belong to mRNA, where U is used instead of T. It also flags the formulation “DNA is translated” as a misconception risk, because DNA is first transcribed to mRNA and mRNA is then translated. This example shows how the feedback output is structured: the agent assigns a severity label, explains the problem, and provides a refinement prompt that can be used by the apply-feedback step.

Not every revision is this substantial. In another exercise, the teacher feedback agent marks the output as a minor revision. The exercise concerns model organisms. The problem statement mentions bacteria, flies, and mice, while the list used in the code also contains *C. elegans*. The feedback points out that *C. elegans* is a worm and is therefore not represented in the introductory sentence. The proposed correction is limited to adding “wormen” to the list of examples. This does not affect the programming task, the solution, or the test cases. However, it improves the consistency between the biological explanation and the data used in the exercise F.3.1.

The student perspective detects a different type of issue. In one case, the student feedback agent flags the explanation that a string can be treated as a “list of letters”. This formulation may confuse beginners because a Python list is a specific data type. The suggested revision is to explain that the program goes through the string “letter by letter”. This keeps the task unchanged, but makes the explanation more precise for students [F.3.1](#).

Contribution to the architecture This iteration introduced a distinct qualitative evaluation and refinement stage. Generation and evaluation therefore became separate functions: the generation component produced a draft, while feedback agents assessed properties such as scientific precision, clarity, and consistency that could not be verified through execution alone. Their output could then be passed to a dedicated refinement component rather than requiring complete regeneration.

Remaining limitation remaining limitation concerned the test cases. Although they could execute successfully with the model solution, this did not guarantee that they were useful for evaluating student code. In one example, the test recreated the solution logic internally and then checked the resulting output. Other tests introduced variables that were not included in the task description or starter code. Such tests were internally executable but insufficiently aligned with the exercise presented to students. [appendix F.3.2](#).

These problems suggest that test generation is a separate design problem. The main exercise-generation prompt is already complex because it has to generate many different exercises components. Test cases require a different kind of precision: they need to match the task exactly and remain consistent with the model solution, while providing meaningful checks of student code. This motivates the next iteration, where test cases are generated in a dedicated step after the exercise and model solution have already been created.

4.2.5 Dedicated test generation

In the fifth iteration, the exercise components and model solution were generated first. A separate generation step subsequently created tests for the already-defined task.

Alignment between generated tests and code execution After examining the generated test cases, a recurring pattern became visible in how the model structured them. Rather than forcing the model to follow one exact test structure, the execution environment was adapted to better support the structures that were generated consistently. This improved the compatibility between generated tests and the code runner. [appendix F.4.1](#).

This result shows that dedicated test generation cannot be improved through prompt changes alone. It also benefits from adapting the surrounding execution environment. This indicates that, at least for test-case generation, the generation

component and its validation environment should be developed in tandem rather than as stand-alone parts.

Contribution to the architecture This iteration established that output components with substantially different requirements can benefit from separate generation stages. Test generation therefore became its own component, operating on an already-defined exercise and model solution. It also demonstrated that generation components and the tools used to validate their output should be designed together rather than treated independently.

Remaining limitations. The follow-up external review confirmed that the testing component had improved, but also identified remaining issues in the student-facing exercise design. First, references such as “in the previous exercise” appear too often. These references are useful for making progression explicit, but they become repetitive when they are used in many consecutive exercises.

Second, some input-output examples were too close to the actual task. Although intended to clarify the expected program behavior, they sometimes exposed the exact transformation or operation students were expected to implement. The input-output pairs should therefore demonstrate the required behavior without directly exposing the solution logic.

Third, the transition from scaffold exercises to basic problem-based exercises is too abrupt. The generated sequence moves from strongly guided tasks to more open tasks without enough intermediate support. This indicated that progression should concern not only the programming concepts being introduced, but also the gradual reduction of scaffolding and the interaction between different exercise types.

These findings informed exercise-level adjustments and later evaluation, but did not lead to another system-level development iteration within the scope of the study. The complete notes from the follow-up external review are included in appendix [F.6.2](#).

4.2.6 eTeacher translation layer

The final iteration addressed the translation of the generated exercise to the eTeacher platform. Most exercise components are already available in the finalized structured output. The problem statement, template code, model solution, and metadata are therefore placed into the correct Y ML fields through a deterministic mapping function. This avoids unnecessary regeneration of content that has already passed through the earlier generation, validation, and refinement stages.

The test cases formed the main exception. eTeacher uses a platform-specific test structure, so the test block was generated separately from the finalized exercise and model solution. The final iteration therefore combined deterministic mapping with focused generation for the part that required platform-specific adaptation.

Contribution to the architecture This iteration established a separation between the platform-independent exercise representation and the platform-specific

output representation. Components that could be transferred directly were handled deterministically, while generation was retained only for elements requiring platform-specific transformation. The core generation process could therefore remain independent of the target platform, with platform-specific logic confined to the final translation stage.

This separation also makes adaptation to other platforms more feasible, because most of the generation pipeline can remain unchanged while the final mapping and platform-specific components are replaced. [appendix F.5](#) illustrates the resulting eTeacher Y ML representation, a figure visualizing the deterministic mapping and platform-specific test cases.

Remaining limitation Although the platform-specific test cases could be generated successfully, the associated student feedback was not always sufficiently informative. The generated test names mainly describe the behavior being checked, such as whether the correct codons are returned or whether remaining letters are ignored. Although these names are understandable, they provide limited diagnostic guidance. They generally do not identify the likely source of the error or direct the student toward a relevant element ([appendix F.5](#)). The iteration therefore addresses platform-specific test representation, but not yet the generation of strongly actionable formative feedback. This showed that generating technically valid platform-specific tests does not automatically result in pedagogically useful feedback. The quality of the feedback therefore remains a separate aspect that requires further refinement.

4.2.7 Final exercise-generation layer

Together, these iterations resulted in the final exercise-generation layer described in [Section 3.5.3](#). Across the development process, the architecture evolved from a single direct prompt into a structured pipeline containing a structured intermediate representation, modular prompt construction, context retrieval, separate generation components, qualitative and technical evaluation, targeted refinement, and platform-specific output translation.

The contributions identified during these iterations form the design knowledge that is examined further in Phase 3. The following section therefore evaluates the internal behavior of the final exercise-generation layer before these findings are combined to derive the general generation architecture.

4.3 Internal behavior of the exercise-generation layer (Phase 3)

The final exercise-generation layer was evaluated by examining the consistency of repeated generations and the behavior of its feedback- and execution-based refinement mechanisms. The results are interpreted descriptively and are intended to

characterize how the different components of the pipeline behave rather than to establish the educational quality of the generated exercises.

4.3.1 Consistency of generated outputs

The consistency analysis compared repeated generations from the same teacher-defined and system-generated blueprints. The results show that the degree and type of variation differ across exercise components.

Code consistency The model solutions show moderate similarity across repeated generations. Average solution CodeBLEU was 0.479 for system-generated blueprints and 0.384 for teacher-defined blueprints. Template CodeBLEU is slightly lower, with averages of 0.372 and 0.339, respectively.

The model solution and template code also tended to vary together. The correlation between both components is 0.558 for generated blueprints and 0.683 for teacher-provided blueprints. This is a desirable pattern, because the template code and the model solution are expected to follow the same programming task and implementation strategy. See [appendix G](#)

Problem statement consistency The problem statements vary significantly in wording. This is visible in the low problem statement BLEU scores: 0.279 for generated blueprints and 0.286 for teacher-provided blueprints. However, embedding similarity is high in both conditions, with averages of 0.872 and 0.875.

The combination of relatively low lexical overlap and high semantic similarity suggests that repeated generations often formulate the exercise differently while preserving broadly similar task meaning. The resulting descriptions are therefore not reproduced identically, but the underlying story, instructions, and intended task remain comparatively similar. Some variation of this kind is expected from probabilistic LLM generation.

Metadata consistency The generated concept metadata also showed partial rather than complete overlap across repeated generations. Jaccard similarity varied between Blueprint, with neither input condition consistently producing identical metadata. This indicates that the set of concepts assigned to otherwise similar exercises can vary between runs.

Test-case consistency The test cases show much lower measured consistency. The average test-case CodeBLEU is 0.066 for generated blueprints and 0.125 for teacher-provided blueprints. This shows that the generated test scripts vary strongly according to CodeBLEU. However, this result should be interpreted carefully. Test scripts have a more complex structure than the other generated components.

The test scripts contain several interacting elements, including execution logic, variable injection, output capturing, assertions, and feedback messages. Small differences in how these elements are formulated can strongly affect a token-based metric

such as CodeBLEU. The current analysis is consequently inconclusive with respect to test-case consistency. Future work should investigate a comparison method that better captures the functional or behavioral equivalence of generated test suites.

Interpretation Overall, repeated generation from the same blueprint does not reproduce identical exercises, but substantial similarity remains in several components. This is clearest for the problem statements, where high semantic similarity is retained despite variation in wording. These findings are consistent with the expected variability of LLM output: the same requirements can lead to different realizations of an exercise without necessarily producing a different task. No equivalent conclusion can currently be drawn for the test cases, because the metric used does not adequately capture their functional similarity.

4.3.2 Feedback-agent refinement

At least one feedback-based refinement was applied to 81 of the 150 exercises generated from system-generated blueprints and to 57 of the 150 exercises generated from teacher-defined blueprints. An exercise could receive revision requests from both agents, so the agent counts are not mutually exclusive.

The higher revision frequency for system-generated blueprints should be interpreted cautiously. The later generated blueprints combined more programming concepts and required alignment between a larger number of exercise elements. Because blueprint complexity was not experimentally controlled, the difference cannot be attributed solely to whether the blueprint was generated or teacher-defined.

The refinement step affected the problem statement considerably more often than the model solution. Among the teacher-defined blueprints, the problem description changed in 40 of the 57 refined exercises, while the solution changed in 14. For the system-generated blueprints, the description changed in 76 of the 81 refined exercises and the solution in 19. Detailed results per blueprint and feedback perspective are provided in [appendix H](#).

The manual review showed several recurring correction patterns. The representative examples underlying these categories are included in [appendix H](#).

Alignment between exercise components A common refinement concerned inconsistent values across the exercise components. For example, one generated exercise describes the DNA string GTCCG, while the example uses GTC. Because the template uses a hard-coded loop limit of 6, this mismatch could confuse students and could lead to an error if the example is followed literally. The feedback agents correct this by aligning the example input, expected output, and template around the same DNA sequence.

Scientific precision A second recurring category concerns biological terminology or precision, where informal or inaccurate terms were replaced by terminology more appropriate to the biological context. The teacher feedback agent often corrects

informal or imprecise wording. For example, descriptions that refer to DN components as general “letters” or “building blocks” are changed to more precise terms such as “basen”, “stikstofbasen”, “nucleotiden”, or “DN -sequentie”.

Alignment with the provided scaffold Several refinements corrected instructions that did not match what students were actually required to complete. In one system-generated exercise, the instructions asked students to create a while-loop even though the loop structure was already provided. The refinement limited the task to completing the condition and accumulator logic that remained open in the template

The same pattern appeared in the teacher-defined blueprint. One exercise instructed students to use `len()` even though `len(dna-sequence)` was already fully present in the template. The refinement removed this contradiction so that the instructions reflected the remaining TODO task

Programming clarity and validity The student agent also detected formulations that could lead to an incorrect programming interpretation. For example, “skip three positions” was replaced with “increase the counter by three,” because the original wording could be interpreted as increasing the counter by four because you want to skip 3 elements.

In the teacher-defined condition, one template placed the colon of a while-loop after a comment marker. This made the template syntactically invalid. Refinement repositioned the TODO structure so that students could complete the condition without accidentally removing or commenting out the colon

The agents also replaced labels such as “input” when the sequence was already stored in a variable, because the term could incorrectly suggest that students needed to call `input()`

Interpretation The reviewed refinements show that the feedback agents mainly corrected local but educationally relevant problems. The teacher agent most often addressed terminology and contextual precision, while the student agent more often identified ambiguity, misleading examples, and mismatches between the instructions and scaffold. When both agents identified related issues, the merged refinement could align several components simultaneously.

The findings also show why execution alone is insufficient for evaluating generated exercises. A model solution may execute correctly even when the example is inconsistent, the biological explanation is imprecise, or the instructions ask students to complete code that is already provided. Feedback agents therefore complement technical validation by evaluating aspects that require interpretation

4.3.3 Execution-based refinement

Execution provided a separate check on the technical compatibility between the model solution and the generated tests. For each generated exercise, the model solution and its test suite were executed together. When this combined execution

failed, the system attempted to refine the solution, the test cases, or both. Table ?? summarizes the initial execution results and the outcome after refinement. Further execution and refinement metrics are provided in appendix I.1.

For the system-generated blueprints, 94 of 150 runs passed the initial execution. Refinement resolved 24 of the 56 initially failing runs, resulting in a final pass rate of 118 of 150. For the teacher-defined blueprints, 127 of 150 runs passed initially, and 18 of the 23 failing runs passed after refinement, resulting in a final pass rate of 145 of 150.

The qualitative review of refined executions included 24 system-generated and 17 teacher-defined cases. Changes occurred mainly in the test cases, while changes to model solutions were less frequent and generally local. Solution changes included adding a missing variable, adapting a loop condition, or removing a hard-coded value. Changes to test cases were often broader because the execution or verification logic itself had to be adapted.

Two different outcomes were observed after test refinement. In several cases, the original testing objective was preserved while an execution problem was corrected. Examples included adapting the test to the execution environment, correcting an expected value, or ensuring that alternative test values were actually used (see appendix I.2.1). In other cases, the test passed because part of the original verification was removed or replaced by a simpler test case. For example, one refinement removed the validation of the program’s printed output and retained only a check of the final variable value. The revised test could therefore execute successfully without verifying whether the student had produced the required output. appendix I.2.3, I.2.2, I.2.4

Remaining failure patterns The reviewed failures showed recurring problems involving mismatches with the execution environment, mismatches between the assumptions of the tests and the structure of the model solution, errors within the generated tests themselves, and cases where refinement weakened the original test. These patterns are summarized in Table 4.2.

4.3.4 Interpretation of Feedback and execution behavior results

The feedback and execution analyses show that the two evaluation mechanisms address different types of problems in the generated exercises. Execution can directly identify technical incompatibilities between the model solution, test suite, and execution environment. It cannot, however, identify problems such as imprecise biological terminology, ambiguous instructions, or a mismatch between the problem statement and the provided scaffold. These were instead recurring targets of the LLM-based feedback agents.

The results therefore support the separation between interpretive and directly verifiable evaluation within the generation architecture. This corresponds to the distinction made in Section 2.3.4, where tool-based validation is proposed for properties that can be checked through execution or deterministic procedures, while

TABLE 4.2: Observed failure patterns in refined executions.

Problem		What happened	Why it matters
Execution-environment mismatch	mis-	The test case expected access to output, code, or variables that the environment did not provide.	correct solution may fail.
Test-solution mismatch	mis-	Test cases expect a different structure for the model solution.	Prevents some tests from actually executing.
Error in the test case		The test case used a wrong expected value, a missing variable, or invalid code.	correct solution could be marked as wrong.
Refinement weakened the test case		Refinement removes or bypasses part of the test.	The test passes while checking less.

LLM-based evaluators are used for properties that require contextual or pedagogical interpretation. The observed refinements show that both mechanisms identify issues that would remain undetected if only the other mechanism were used.

t the same time, the execution analysis also demonstrates a limitation of automatic refinement. Resolving an execution failure does not necessarily mean that the resulting test is better: in some reviewed cases, refinement produced a passing test by weakening the original verification. One possible improvement is to provide the test-generation agent with clearer information about the execution environment and with concept-specific testing guidance that defines the intended behavior and common student mistakes. The refinement agent could also receive clearer instructions to preserve these testing objectives while resolving execution errors. But this should be explored more in the future. More broadly, LLM-based test-case generation remains an active research area with recurring challenges related to correctness, executability, coverage, and reliability[2]. The design and evaluation of this component should therefore be explored further in future work.

4.4 General generation architecture (*Phase 3*)

The development of the exercise-generation layer resulted in the general generation architecture shown in Figure 4.2. The architecture describes a reusable structure for controlled LLM-based generation. It separates input preparation, context selection, generation, evaluation, refinement, and output storage into distinct components. The exact implementation of these components can vary according to the generation task.

The architecture starts from a structured input, such as the input schema defined in Section 3.3 for exercise generation. Before generation, this input can be enriched with information selected from a context library. The context component determines which external knowledge or previously generated outputs are relevant to the current

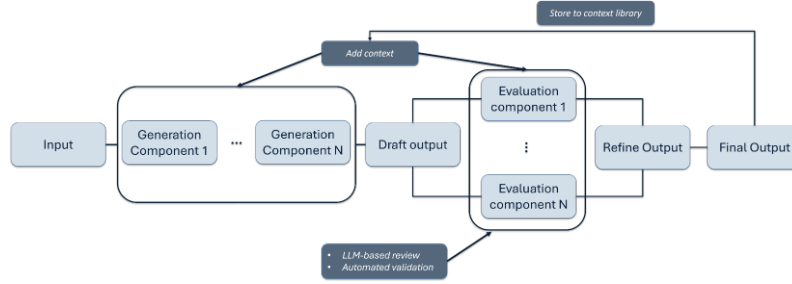


FIGURE 4.2: General architecture

task and adds them to the generation input. This allows the generator to use domain information and earlier outputs without placing all available context in every prompt.

Generation components The prepared input is processed by one or more generation components. A task can use a single generation step when its output can be produced coherently within one call. Additional components can be introduced when parts of the output require different instructions, input information, or validation.

The separation of test-case generation in Section 4.3.3 illustrates this principle. Tests require a different focus from the main exercise content and depend on an already defined task and solution. The general architecture therefore supports a variable number of generation components rather than prescribing one fixed pipeline.

The components produce a structured draft. This draft combines the generated artifacts in an intermediate representation and forms the boundary between generation and quality control. The output is not considered final immediately, but remains available for evaluation and targeted modification.

Evaluation components The draft is assessed by one or more evaluation components. These components are defined according to the properties that need to be evaluated and are therefore not limited to the feedback agents used in the exercise-generation layer.

Some properties can be checked through deterministic procedures or external tools. Others require an interpretive assessment by an LLM-based evaluator. The architecture allows these mechanisms to be combined while keeping their responsibilities separate.

The results in Sections 4.3.2 and 4.3.3 show the importance of this distinction. Code execution detects technical incompatibilities but cannot determine whether terminology is precise or whether the instructions match the provided scaffold. Conversely, an LLM-based evaluator can identify such qualitative issues but is less reliable for properties that can be verified directly through execution. The architecture therefore assigns different evaluation properties to suitable evaluation mechanisms.

Refinement and finalization When an evaluation component identifies an issue, its output is passed to the refinement component. Refinement modifies the existing

draft based on the identified problems rather than restarting the complete generation process. This preserves suitable parts of the output while allowing specific components to be corrected.

In the implemented architecture, evaluation and refinement are performed once. The refined result is subsequently accepted as the final output. During development, this approach produced acceptable results for the reviewed teacher-defined exercises, and an additional evaluation pass was not considered necessary. A single pass also limits token consumption and execution time.

The broader repeated-run evaluation nevertheless exposed cases in which test refinement resolved an execution failure by weakening part of the original verification. This suggests that future versions should investigate an iterative refinement loop in which revised output can be evaluated again. Such a loop is one possible improvement alongside stronger test-generation constraints, clearer execution-environment information, and more explicit refinement instructions. Its additional quality benefit would need to be balanced against the corresponding computational and token costs.

Final output and context reuse After refinement, the result becomes the final structured output. Depending on the application, this output can be translated to a target format, stored, presented to a user, or passed to another system.

The final output can also be added to the context library. This creates the feedback path shown in Figure 4.2: finalized outputs can become contextual input for later generation tasks. The architecture can therefore support both isolated generation and sequential processes in which later outputs build on earlier ones.

Generality of the architecture Although both layers follow the same general functional structure, the individual components are adapted to the artifact being generated. Table 4.3 compares how these components are implemented in the exercise- and sequence-generation layers. The comparison shows that the architecture does not prescribe one fixed context mechanism or set of feedback agents. These are selected according to the generated artifact and the properties that need to be assessed.

Applying the same structure to both layers provides preliminary evidence that the architecture can be reused for related generation tasks within the same educational context. In particular, the sequence-generation layer applies the structure at a different level of the generation process, where the generated blueprints are evaluated in relation to their role within an intended progression rather than as complete exercises. This suggests that the architecture can support different educational generation purposes while retaining the same underlying functional organization. However, this generality has so far only been demonstrated for these two closely related implementations.

General component	Exercise-generation layer	Sequence-generation layer
Input	Exercise blueprint	Learning goal
Context mechanism	Retrieved domain knowledge and earlier exercises	Sequence initialization and sliding window of earlier blueprints
Generation	Exercise content and dedicated test generation	Sequential blueprint generation
Evaluation	Teacher- and student-oriented feedback agents, code execution, and deterministic checks	Context, prior-knowledge, and progression feedback agents, supported by deterministic checks

TABLE 4.3: Comparison of how the general architecture is implemented in the exercise- and sequence-generation layers.

4.4.1 Architectural contribution and scope

The main architectural result is therefore not a fixed pipeline for generating programming exercises, but a reusable functional structure for controlled LLM-based generation. In the context of this thesis, this structure was developed to keep the teacher in control of the educational choices that guide the generation process, while leaving the translation of these choices into prompts, contextual information, generation steps, and evaluation procedures to the system.

Several elements of this approach can also be found in existing educational generation systems. Intelli similarly allows instructors to provide structured pedagogical choices that are translated into prompts by the system, reducing the need to construct complete prompts manually[36]. ExGen adds an evaluation step that filters generated exercises based on their assessed difficulty[33]. The architecture developed here extends this principle by using evaluation not only to determine whether an output is suitable, but also to identify specific problems that can guide targeted refinement. PyTaskSyn, in turn, demonstrates how different agent perspectives can be used during generation and evaluation[27]. A similar principle is used here through feedback agents that assess the generated material from different educational perspectives. The architecture brings these mechanisms together within a generation process in which the teacher specifies what should be generated, while different components determine how these requirements are translated into a usable exercise.

The application of the same architecture to the sequence-generation layer indicates how this approach can also support teachers at a different level of the exercise-creation process. Rather than starting from an already defined exercise blueprint, sequence generation can start from broader sequence requirements and produce the blueprints that guide the subsequent generation of individual exercises. This suggests that the general architecture can support different generation tasks involved in creating programming material, although the extent to which these tasks should be automated remains an open question.

Together, these results provide an answer to RQ1 by showing that contextualized programming exercises for classroom use can be generated through a controlled process in which teachers define the educational intent while the system supports the translation of that intent into generated and evaluated material. The results also point towards a broader question for future work: which parts of the generation process should be handled by the system, and where should teachers remain directly involved? Future work should therefore investigate how different implementations of the architecture can interact, how a human-in-the-loop component can be incorporated, and how these tools can be integrated into teachers' existing workflows without introducing additional burden. The current implementations provide preliminary evidence that the architecture can support this type of assisted generation, but further evaluation is needed to determine how broadly and effectively it can be applied.

4.5 Teacher and domain-expert evaluation *Phase 4*

The teacher and domain-expert evaluation examined the generated sequences from complementary perspectives. Teachers focused mainly on classroom use and student learning, whereas domain experts focused more strongly on programming-related quality, progression, terminology, solutions, and test cases. Because of the small number of participants, the findings are interpreted descriptively and are organized according to the evaluation dimensions defined in Section 3.3. [appendix J](#)

Technical and structural quality The domain experts generally considered the generated exercises technically plausible, and the model solutions usually implemented the requested tasks. However, several remaining inconsistencies were identified, including issues with terminology, formatting, concept naming, scaffolds, and the alignment between exercise components. Test cases were a more notable limitation. Some tests focused too strongly on exact output, uncommon edge cases, or specific implementation details instead of the intended programming concept. This corresponds with the internal evaluation, where test-case generation and refinement were also found to be less reliable than other components.

The teacher perspective complemented these findings. The feedback linked to the test cases was evaluated less consistently, suggesting that the generated feedback does not always help students understand their mistakes sufficiently. This indicates that test generation should be more closely aligned with the learning objective of the exercise. One possibility is to give teachers more control by allowing them to specify which behaviors or common student mistakes should be tested. A second possibility is to improve the test-generation component itself by incorporating additional knowledge about educational test design, either as contextual information provided to the test-generation agent or through more explicit test-generation instructions. This could help ensure that generated tests focus on the intended programming concept and provide feedback that better supports student learning. This result corresponds with our previous finding of phase 2 and 3.

Pedagogical quality The main pedagogical concern was not that the exercises lacked structure, but that they were often insufficient as a first introduction to a programming concept. Both teachers repeatedly indicated that students would benefit from seeing explanations or examples before being asked to apply a new construct. Their open responses specifically suggested replacing part of the written explanation with examples or visualizations and reducing the amount of information contained in each exercise. The interview questions and notes similarly focus on the need for different activity types, examples, and stronger integration of the exercises into teacher-led instruction.

Domain experts identified related issues from a programming perspective. Several comments point to inconsistencies in terminology, formatting, scaffolds, and concept naming. Some concepts are introduced unclearly or with names that may not be meaningful for students. For example "string slicing" In some cases, scaffolds also contain elements that are not part of the intended concept progression.

Contextual quality The biological context itself was not identified as the main weakness of the generated material. Some expert feedback explicitly described exercises as more interesting when the programming task was more closely connected to the biological context. However, the teacher responses regarding whether the context distracted from programming were mixed.

much clearer recurring finding concerned the amount of information in the problem statements. Both teachers consistently reported that the exercises contained too much text, with open responses repeatedly calling for shorter descriptions, more examples, and visual representations. The problem therefore appears to concern how much contextual and instructional information is presented at once rather than contextualization alone

Teacher usability and adaptation The teacher evaluation indicates that the generated material would still require adaptation before classroom use. The reported effort varied between sequences, from small to substantial adaptations. However, the qualitative responses show that this effort does not mainly consist of rewriting technically incorrect exercises. Teachers repeatedly referred to shortening the text, adding examples or visualizations, introducing concepts before students use them, and deciding how the exercises should be incorporated into the lesson.

The generated sequences should therefore be viewed more as material that supports lesson preparation than as complete standalone lessons. This also clarifies the role of teacher control. Teachers may want to determine not only the target concepts and exercise types, but also how exercises are combined with explanations, examples, or other learning activities.

Interpretation Taken together, the evaluation suggests that the generated exercises are a useful starting point, but that quality at the level of individual exercise components is not sufficient for classroom readiness. Domain experts mainly identified remaining issues in test design, terminology, scaffolding, and progression, while

teachers focused more on how the material would be introduced and used during instruction.

These limitations correspond closely with the design conditions identified in the literature. Introductory programming exercises should remain focused on a clear learning goal, introduce new demands gradually, provide appropriate scaffolding, and give feedback that helps students understand their mistakes. The literature also warns that overly dense problem statements and excessive supporting information can increase cognitive demands. The current generation approach already incorporates several of these principles through structured inputs, sequence-aware generation, and evaluation components, suggesting that the system is moving in the right direction. However, the evaluation shows that these requirements are not yet controlled consistently enough in the generated output.

The main implication is therefore that the system should support teacher-led lesson construction rather than only generate complete exercise sequences. This includes allowing variation between explanations, examples, scaffolded tasks, and more independent exercises, while preserving teacher control over how these are combined. The amount of information should also be managed more carefully: information required to solve the exercise should remain directly available, while additional explanations or contextual information could be presented separately or on demand.

Test generation remains a separate priority. The findings indicate that tests should be more closely aligned with the intended programming concept and with the type of feedback that is useful for beginners. This could be supported through stronger test-generation instructions, additional educational testing knowledge in the generation context, and teacher control over what behaviors or misconceptions should be assessed.

Overall, this answers RQ2 by suggesting that the generated material is promising but not fully classroom-ready. The main remaining need is not simply better generation of individual exercises, but better support for how teachers assemble, present, and adapt the generated material within their instructional workflow.

4.6 Classroom study

4.6.1 Comparative motivation findings

The comparative SIMS analysis did not reveal a consistent motivational effect of contextualization. Before the intervention, intrinsic motivation was already higher in the contextualized group at the exploratory significance level of $\alpha = .1$ ($p = .64$). After the intervention, motivation was higher in the control group ($p = .21$), while intrinsic motivation increased within the control group ($p = .61$). No significant pre-post changes were found within the contextualized group. The complete statistical analysis is reported in the unpublished Educational Master's study [6].

These findings do not provide clear evidence that biological contextualization increased motivation. The baseline difference between the groups and the contrasting

4. RESULTS AND DISCUSSION

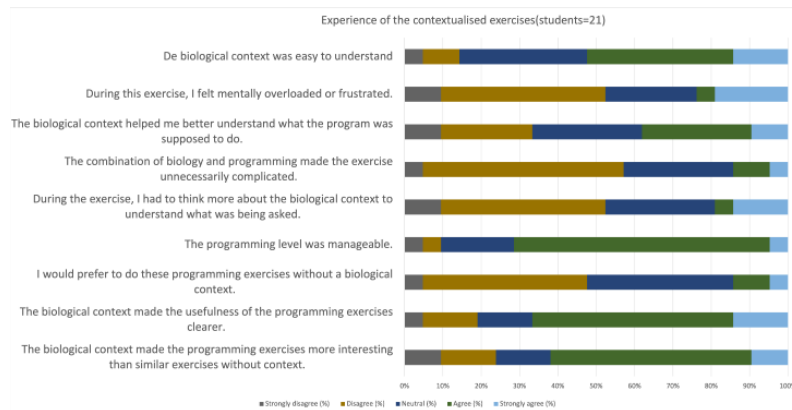


FIGURE 4.3: The results on how students experienced the contextualized exercises

changes in the control group limit the interpretation of the post-intervention difference in amotivation. The remainder of this section therefore focuses on students' direct evaluation of the contextualized exercises.

4.6.2 Experience of the contextualized exercises

Figure 4.3 presents the response distributions for the nine questionnaire items. The overall evaluation of the biological context was positive. Most students indicated that the context made the exercises more interesting and clarified the usefulness of programming. Only a small proportion preferred completing the exercises without a biological context.

The programming level was also generally perceived as feasible. Most students did not consider the combination of biology and programming unnecessarily complicated and did not report mental overload or frustration. This suggests that the biological context did not create excessive additional difficulty for most students.

While most students indicated that the context clarified the usefulness of programming, responses were mixed on whether it helped them understand what the program should do. This suggests that, in this exercise set, the connection between the biological situation and the required program behavior was not always sufficiently explicit.

The response distributions also indicate that the context was not evaluated equally by all students. While many students experienced it as interesting and useful, others remained neutral or perceived greater difficulty. The same contextualization may therefore support some students while adding little value, or even additional difficulty, for others. This is consistent with the theoretical discussion in Section 2.2, where contextualization is treated as dependent on the relationship between the learner, the context, and the task rather than as an inherently beneficial exercise characteristic[23, 21].

4.6.3 Selected exploratory associations

Only correlations that contribute directly to interpreting the role of contextualization are discussed here. The complete correlation matrix is included in appendix ??.

Perceived usefulness was positively associated with post-intervention identified regulation, $\rho = .48$, $p = .29$. Students who reported that the biological context made the usefulness of programming clearer also tended to show higher identified regulation.

Perceived usefulness was also positively associated with the pre-post change in identified regulation, $\rho = .52$, $p = .16$. Students who evaluated the context as making programming more useful therefore tended to show a more positive change in the extent to which they recognized the value of programming.

This relationship is consistent with the theoretical role of identified regulation, which concerns participating in an activity because its importance or usefulness is recognized. However, the association does not establish that the contextualization caused the motivational change. Students who already valued programming may also have evaluated the context more positively.

second relationship concerned the functional role of the context. Students who reported that the biological context helped them understand what the program should do also tended to find the exercises more interesting, $\rho = .673$, $p < .1$. This suggests that contextualization was more engaging when it contributed directly to understanding the programming task rather than functioning only as a surrounding story.

Perceived programming feasibility was strongly negatively associated with unnecessary complexity, $\rho = -.741$, $p < .1$. Students who considered the combination of biology and programming unnecessarily complicated also tended to regard the programming level as less manageable. Although this result does not distinguish between difficulty caused by the programming content and difficulty caused by the biological context, it shows that additional contextual complexity was closely related to students' overall perception of task feasibility.

4.6.4 Interpretation and design implications

The results suggest that the value of contextualization depends on how well the biological context is integrated into the programming task. The biological framing was more consistently perceived as making the usefulness of programming visible than as helping students understand what the program should do. For this exercise sequence, the generator therefore appears to have conveyed contextual relevance more successfully than the operational connection between the biological situation and the required programming behavior.

This distinction is consistent with previous research on contextualized programming problems, which shows that contextualization can increase perceived relevance or authenticity, but may also introduce additional cognitive demands when the context is unfamiliar or insufficiently aligned with the programming task [23, 21].

The strong relationship between perceived programming feasibility and unnecessary complexity suggests that students who experienced more difficulty with the programming task also tended to experience the combination with biology as more demanding. The current data cannot determine whether this originated from the programming concepts, the biological context, or their interaction. The first refinement is therefore to inspect the exercises in which both demands coincide and determine whether the context, instructions, or programming requirements can be simplified.

The second direction is to explore greater differentiation in context selection, for example by offering alternative contexts or allowing students to choose between them. Since students' attitudes toward biology were not measured, this remains a possible explanation rather than a conclusion.

Regarding RQ3, students generally experienced the generated contextualized programming exercises positively, although this experience differed between students. Most students perceived the biological context as making the exercises more interesting and the usefulness of programming more visible, while its contribution to understanding the required program behavior was less consistent. The comparative analysis did not provide clear evidence that the contextualized exercise sequence increased motivation at group level. However, students who perceived the context as making programming more useful tended to report higher identified regulation and a more positive change in identified regulation. The perceived motivational impact of contextualization therefore appears to lie mainly in making the value of programming more visible for some students, rather than in producing a uniform increase in motivation across the group.

4.6.5 Limitations

The findings are exploratory and based on a small contextualized group. The questionnaire measures students' perceptions rather than direct learning outcomes, and the correlations do not establish causal relationships. The change-score correlation is also sensitive to baseline values and measurement variation.

In addition, students' initial interest in biology, biological prior knowledge, and biological self-efficacy were not measured. It is therefore not possible to determine whether these characteristics explain why students evaluated the same context differently.

4.7 Conclusion

This thesis investigated whether LLM-based generation can support the creation of programming exercises tailored to the BCSW study track. The problem motivating this work is that suitable programming material for this relatively small educational context is limited, while creating and adapting interdisciplinary exercises manually requires substantial teacher effort. At the same time, contextualized exercises must satisfy more than technical correctness: they need to align with the intended programming goals, use the biological context meaningfully, match students' prior knowledge, and remain usable within classroom practice.

The study therefore developed and evaluated an LLM-based generation approach through an iterative design-based process. The resulting system was examined from three complementary perspectives: how the generation process can be structured and controlled, how teachers and domain experts evaluate the generated material, and how students experience the contextualized exercises. Together, these perspectives provide the following answers to the research questions.

RQ1: How can LLMs generate interdisciplinary programming exercises?

The development process suggests that interdisciplinary programming exercise generation can benefit from being approached as a controlled generation pipeline rather than as a single prompt-to-output task. The final architecture separates structured input, contextual information, generation, evaluation, refinement, and output formatting. Directly verifiable properties, such as code execution and output structure, can be evaluated through deterministic tools, while LLM-based feedback can be used to assess properties that require interpretation, such as clarity, contextual appropriateness, and pedagogical alignment. Generated material is therefore treated as a draft that can be evaluated and refined before becoming final output.

A central design principle that emerged is the separation of concerns. Different parts of the generated artifact have different requirements and may benefit from different generation and evaluation mechanisms. This was particularly visible for test cases, for which dedicated generation and execution provided greater control than generating them as another part of one large prompt.

The internal evaluation further illustrates both the potential and limitations of these mechanisms. Repeated generations often preserved similar overall meaning while varying in their exact wording and code structure. The feedback and execution-based refinement mechanisms corrected a number of observed issues, but did not do so reliably in every case. In particular, some test refinements weakened the test or removed the immediate cause of an execution failure without resolving the underlying problem. Automated refinement should therefore be interpreted as an additional quality-control mechanism rather than as a guarantee of correctness.

The recurring functional structure of the exercise-generation layer was abstracted into a more general generation architecture. A similar structure was subsequently used for sequence generation: structured requirements are provided as input, relevant context is maintained between generation steps, blueprints are generated and evaluated, and the resulting structured output becomes input to the exercise-generation layer. This provides an initial indication that the architecture can be applied to more than one related educational generation task. However, the exercise-generation layer received substantially more internal evaluation than the sequence-generation layer, so the findings do not provide the same level of support for both implementations.

RQ2: How do teachers and domain experts evaluate quality and usability?

The teacher and domain-expert evaluation suggests that the generated material can provide a useful starting point for educational content, while also in-

dicating that the quality of individual programming exercises and their usability within classroom instruction should be considered separately.

Within the small domain-expert evaluation, the programming-related quality of the material was generally assessed positively. Model solutions were considered appropriate, progression was largely viewed as gradual, and the sequences were generally considered to correspond with their intended programming goals. At the same time, the evaluations identified remaining issues in areas such as test-case focus, edge cases, terminology, scaffolding, and repetition between exercises.

The teacher evaluations placed more emphasis on how the material would fit into instruction. The biological context itself was not identified as the main difficulty. Instead, recurring concerns involved the amount of information presented, scaffolding, and the role of the exercises within a lesson. The evaluated sequences were not considered sufficient as standalone introductions to new programming concepts, and the teacher feedback indicated a need for a broader range of instructional activities, including explanation, examples, scaffolded practice, and more independent programming tasks.

Taken together, these perspectives suggest that technically and pedagogically plausible programming material may still require adaptation before it fits a specific classroom setting. The small number of participants prevents broad conclusions about teacher or domain-expert preferences, but the recurring observations indicate that teacher mediation remains important. The generated exercises should therefore not be considered complete standalone lesson material, but rather material that still needs to be selected, adapted, and integrated into the teacher's broader instructional approach.

These findings also suggest that the purpose of the generator should not be to remove the teacher from the generation process. Instead, it can reduce the amount of material that needs to be created from scratch while preserving teacher control over selection, ordering, adaptation, and classroom integration.

RQ3: How did students experience the generated contextualized programming exercises, and what was their perceived impact on motivation?

The classroom study provides a generally positive but non-uniform picture of how students experienced the generated exercises. Most students in the contextualized group reported that the biological context made the exercises more interesting and made the usefulness of programming clearer. Most also considered the programming level feasible and did not report that combining biology and programming created unnecessary complexity or excessive mental overload. However, students differed in how positively they evaluated the context, and responses were more mixed regarding whether the biological context helped them understand what the program itself was supposed to do.

The exploratory associations suggest that perceived usefulness of the context may be relevant to students' motivational experience. Students who reported that the biological context made the usefulness of programming clearer tended to show higher identified regulation after the intervention, and perceived usefulness was positively

associated with the pre–post change in identified regulation. Students who felt that the context helped them understand the required program also tended to evaluate the exercises as more interesting. These associations are consistent with the idea that contextualization may be more valuable when it contributes functionally to the programming task, but they do not establish a causal relationship.

The comparative motivation results do not provide clear evidence that the contextualized exercises caused an increase in motivation. Baseline differences between the groups and the absence of significant pre–post changes within the contextualized group prevent such a conclusion. The results instead suggest that, for at least part of the group, contextualization may help make the relevance and usefulness of programming more visible.

Overall, the classroom results suggest that the architecture was able to generate contextualized exercises in which the biological context made the relevance of programming more visible to many students. However, this connection was not experienced uniformly. The generated context appeared to communicate the usefulness of programming more consistently than it clarified the required programming behavior.

Main contribution The main contribution of this thesis is the resulting generation architecture, which provides a starting point for developing LLM-based educational generation engines that go beyond generating isolated standalone exercises. The architecture structures generation around explicit educational requirements, contextual information, generation, evaluation, and refinement, allowing different parts of the process to be controlled and adapted.

Rather than aiming for fully autonomous generation, the architecture is intended to support educators in creating material that fits within a broader instructional process. The generated output can serve as input for subsequent generation steps, be combined into sequences, and be reviewed or adapted before classroom use. In this way, control remains with the educator over what is generated, how material is organized, and how it is ultimately integrated into teaching.

The thesis therefore provides a foundation for further development of generation engines in which LLMs support the creation of educational material while the structure of the system preserves educator control over the design and use of that material.

4.8 future work

The results of this thesis point to several directions for further development of both the generation architecture and its evaluation. The most relevant next steps concern the testing component, more systematic evaluation during system development, integration into the teacher workflow, and further investigation of how students experience the generated material.

Improving test-case generation and feedback Test generation remains an important area for refinement. Future work should define the target execution environment more explicitly and improve the alignment between the intended programming behavior, the generated tests, and their feedback. Generating an explicit test plan before the final tests could help specify which behaviors and common mistakes should be checked. The associated feedback could also be evaluated separately for whether it is understandable and actionable for beginning programmers.

More systematic evaluation during system development The evaluation performed on the final exercise-generation layer could be integrated more directly into future development iterations. Rather than using repeated-run and refinement analyses mainly to evaluate a completed version, similar evaluations could function as a recurring benchmark whenever a generation, evaluation, or refinement component is modified. This would make it easier to determine whether a change improves the intended component without introducing problems elsewhere in the pipeline.

Integration into the teacher workflow The next step is to develop a teacher-facing interface and evaluate the architecture during actual lesson preparation. The current teacher and domain-expert evaluations required participants to assess complete sequences through relatively extensive questionnaires. This provided useful feedback, but it also placed a considerable evaluation burden on participants and only captures their judgment after the material has already been generated.

Integrating generation and evaluation into an interface used during actual lesson preparation would allow the system to be studied in a more natural setting. Future studies could examine which generated exercises teachers retain, modify, reorder, or reject, providing more direct evidence of the required adaptation effort and the controls teachers need. These interactions could provide more direct evidence of where the generator reduces teacher effort and where adaptation is still necessary.

The system could also support greater variation in exercise and activity types and allow teachers to control how much supporting or contextual information is presented. Future development should therefore investigate not only the quality of individual generated artifacts, but also how teachers and a generation tool can work complementary.

Further evaluation from the student perspective The classroom study provides a first basis for identifying how students experience the generated contextualized exercises, but the differences between students indicate that this perspective should be investigated further. Additional classroom studies could investigate which characteristics of the exercises generated by the architecture contribute to different student experiences. These findings could then be used to improve the student-perspective evaluation component of the generation architecture. Rather than relying primarily on general pedagogical criteria, this evaluator could increasingly incorporate empirical findings about how the target students actually experience the generated exercises.

ppendices

ppendix

Exploratory Study Instruments

.1 Teacher Survey

. EXPLORATORY STUDY INSTRUMENTS

Een verkennend onderzoek naar informatica-leerdoelen binnen BCSW

* Required

Introductie

Dit formulier kadert binnen een onderzoek van de KU leuven naar het realiseren van de **informatica-doelstellingen** in de derde graad secundair onderwijs, specifiek binnen de studierichting **Biotechnologische en Chemische STEM-wetenschappen (BCSW)**.

Het doel van deze vragenlijst is om de **huidige onderwijspraktijk en context** in kaart te brengen: hoe informaticadoelen worden aangebracht, welk lesmateriaal wordt gebruikt, **hoe leerlingenmotivatie wordt ervaren**, en in welke mate programmeeronderwijs wordt gekoppeld aan vakinhouden zoals biologie.

De resultaten van deze bevraging vormen **een eerste verkennende stap** en zullen gebruikt worden om in een volgende fase **ondersteunend lesmateriaal en programmeeroefeningen te ontwerpen en te verfijnen**.

Uw antwoorden worden **anoniem verwerkt** en uitsluitend gebruikt voor onderzoeksdoeleinden. Er zijn geen juiste of foute antwoorden; we zijn vooral geïnteresseerd in uw ervaringen en inzichten als leerkracht.

Het invullen van deze vragenlijst duurt ongeveer **10-15 minuten**. U kan op elk moment stoppen indien u dat wenst.

Contact: Heeft u vragen, dan kan u mij steeds contacteren via: gino.vancleemput@student.kuleuven.be

Achtergrond leerkracht

In deze eerste sectie verzamelen we enkele algemene gegevens over uw **leservaring en achtergrond**. Deze informatie helpt ons om de antwoorden in de rest van de vragenlijst beter te kunnen interpreteren.

Voor welke vakken geeft u momenteel les in de derde graad? *

- ☐ Biologie
- ☐ Chemie
- ☐ Informaticawetenschappen/equivalent
- ☐ Other

Hoeveel jaar ervaring heeft u met lesgeven? *

- ☐ Minder dan 2 jaar
- ☐ 2-5 jaar
- ☐ 6-10 jaar
- ☐ Meer dan 10 jaar

In welke mate beschikt u over voorkennis of bijkomende opleiding voor het vak informaticawetenschappen/equivalent? *

- ☐ Geen formele opleiding
- ☐ Basis via lerarenopleiding
- ☐ Nascholingen / bijscholingen
- ☐ Zelfstudie

Als u dit wenst, mag u de bovenstaande vraag wat meer toelichten (optioneel).

74 Organisatie van de informatica-leerdoelen binnen de school

In deze sectie focussen we op hoe de **informatica-leerdoelen** binnen uw school worden georganiseerd en aangeboden. We kijken naar de plaats van deze leerdoelen binnen het curriculum en naar eventuele samenwerking tussen leerkrachten.

.1. Teacher Survey

Op welke manier worden de informaticadoelen op uw school aangeboden? *

- ☐ Als een apart vak informaticawetenschappen
- ☐ Geïntegreerd in andere vakken (bv. STEM, wetenschappen)
- ☐ Zowel apart als geïntegreerd
- ☐ Other

Hoe wordt het gegeven curriculum verspreid over de tijd? *

- ☐ Gelijk doorheen het hele jaar
- ☐ Enkel in het eerste semester
- ☐ Enkel in het tweede semester
- ☐ Other

Welke rol neemt u zelf op in het realiseren van de informaticadoelen op uw school? *

- ☐ Lesgever
- ☐ Co-teacher
- ☐ Vakcoördinator
- ☐ Other

In welke mate wordt er samengewerkt tussen leerkrachten van verschillende vakken bij het aanbrengen van informaticadoelen? *

1 = helemaal niet → 5 = zeer intensief

	1	2	3	4	5
Hoe vaak	☐	☐	☐	☐	☐

. EXPLORATORY STUDY INSTRUMENTS

Terminologie

In de rest van deze vragenlijst gebruiken we:

de term “**informaticawetenschappen/equivalent**” als verzamelterm voor het **aanleren van informatica-leerdoelen**, ongeacht of deze worden aangeboden als een apart vak of geïntegreerd binnen andere vakken (zoals STEM of wetenschappen). *

☐ Gelezen

.1. Teacher Survey

Motivatie van leerlingen

In dit onderdeel peilen we naar uw **ervaringen met de motivatie en interesse van leerlingen** voor informaticawetenschappen.

Hoe ervaart u de motivatie van leerlingen voor het vak informaticawetenschappen/equivalent? *

1 = totaal niet gemotiveerd → 5 = zeer gemotiveerd

	1	2	3	4	5
Motivatie	☐	☐	☐	☐	☐

Merkt u verschillen in motivatie of interesse tussen verschillende studierichtingen (voor het vak informaticawetenschappen/equivalent)? *

1 = totaal geen verschil → 5 = zeer groot verschil

	1	2	3	4	5
Vershil	☐	☐	☐	☐	☐

Welke factoren lijken volgens u dit verschil te veroorzaken? (optioneel)

. EXPLORATORY STUDY INSTRUMENTS

Gebruik van lesmateriaal en digitale platformen

Deze sectie gaat over het **lesmateriaal** dat u gebruikt voor informaticawetenschappen/equivalent. Afhankelijk van uw situatie volgen er vragen over het gebruik van **eTeacher**, een ander digitaal platform, of zelfontwikkeld lesmateriaal.

Maakt u gebruik van eTeacher of een gelijkaardig digitaal platform om informaticawetenschappen/equivalent te geven? *

eTeacher is een online leerplatform met interactieve lespakketten voor het secundair onderwijs waarin leerlingen op eigen tempo leren programmeren met theorie, oefeningen en automatische feedback.

- ☐ Ja, ik gebruik eTeacher
- ☐ Ja, ik gebruik een ander platform (gelijkaardig aan eTeacher)
- ☐ Nee, ik gebruik geen specifiek platform
- ☐ Other

.1. Teacher Survey

eTeacher

De volgende vragen focussen specifiek op uw ervaringen met **eTeacher**.

(Met **context** bedoelen we dat programmeerconcepten worden aangebracht aan de hand van herkenbare situaties of vakinhouden uit andere vakken, zoals biologie)

Sluit de huidige context in eTeacher voldoende aan bij de leefwereld van leerlingen uit BCSW? *

1 = helemaal niet → 5 = volledig

	1	2	3	4	5
Sluit het aan?	☐	☐	☐	☐	☐

Hoe vaak gebruikt u de volgende bronnen voor uw lesmateriaal in informaticawetenschappen/equivalent? *

1 = nooit → 5 = altijd

	1	2	3	4	5
Ik maak mijn eigen les materiaal	☐	☐	☐	☐	☐
Ik gebruik bestaand lesmateriaal dat niet afkomstig is van eTeacher	☐	☐	☐	☐	☐
Ik gebruik lesmateriaal uit eTeacher	☐	☐	☐	☐	☐

. EXPLORATORY STUDY INSTRUMENTS

Een ander digitaal platform

Indien u een ander digitaal platform gebruikt, peilen we hier naar uw ervaringen.

(Met **context** bedoelen we dat programmeerconcepten worden aangebracht aan de hand van herkenbare situaties of vakinhouden uit andere vakken, zoals biologie)

Wat is de naam van dit platform? *

Sluit de huidige context in dit platform voldoende aan bij de leefwereld van leerlingen uit BCSW? *

1 = helemaal niet → 5 = volledig

	1	2	3	4	5
Sluit het aan?	☐	☐	☐	☐	☐

Hoe vaak gebruikt u de volgende bronnen voor uw lesmateriaal in informaticawetenschappen/equivalent? *

1 = nooit → 5 = altijd

	1	2	3	4	5
Ik maak mijn eigen les materiaal	☐	☐	☐	☐	☐
Ik gebruik bestaand lesmateriaal dat niet afkomstig is van dit digitaal platform	☐	☐	☐	☐	☐
Ik gebruik lesmateriaal afkomstig van dit digitaal platform	☐	☐	☐	☐	☐

.1. Teacher Survey

Maakt geen gebruik van een specifiek platform

Indien u geen specifiek digitaal platform gebruikt, focust deze sectie op uw **eigen lesmateriaal** en bestaand materiaal dat u inzet in de lessen informaticawetenschappen.

(Met **context** bedoelen we dat programmeerconcepten worden aangebracht aan de hand van herkenbare situaties of vakinhouden uit andere vakken, zoals biologie)

Sluit de huidige context van u lesmateriaal voldoende aan bij de leefwereld van leerlingen uit BCSW? *

1 = helemaal niet → 5 = volledig

	1	2	3	4	5
Sluit het aan?	☐	☐	☐	☐	☐

Hoe vaak gebruikt u de volgende bronnen voor uw lesmateriaal in informaticawetenschappen/equivalent? *

1 = nooit → 5 = altijd

	1	2	3	4	5
Ik maak mijn eigen lesmateriaal	☐	☐	☐	☐	☐
Ik gebruik bestaand lesmateriaal.	☐	☐	☐	☐	☐

. EXPLORATORY STUDY INSTRUMENTS

Type oefeningen

In deze sectie vragen we u om verschillende **types programmeeroefeningen** te beoordelen. We bekijken in welke mate deze oefeningen volgens u geschikt zijn voor het **aanleren van verschillende informatica-leerdoelen**, zoals basisbegrippen, programmeervaardigheden, datastructuren en algoritmisch denken.

Hoe geschikt vindt u de volgende types oefeningen voor het aanleren van: **Basisbegrippen & programmeerfundamenten**?
(Focus op begincompetenties, basislogica en eerste programmeertaalconcepten.) *

1 = helemaal niet geschikt -> 5 = zeer geschikt

	1	2	3	4	5
Gesloten opdrachten	☐	☐	☐	☐	☐
Opdrachten die stapsgewijs opbouwen naar een groter geheel	☐	☐	☐	☐	☐
Contextopdrachten	☐	☐	☐	☐	☐
Projectopdrachten	☐	☐	☐	☐	☐
Interactieve oefeningen	☐	☐	☐	☐	☐

Hoe geschikt vindt u de volgende types oefeningen voor het aanleren van: **Kern van het Programmeren**?
(Focus op control flow, variabelen, lussen, functies en conditionals.) *

1 helemaal niet geschikt -> 5= zeer geschikt

	1	2	3	4	5
Gesloten opdrachten	☐	☐	☐	☐	☐
Opdrachten die stapsgewijs opbouwen naar een groter geheel	☐	☐	☐	☐	☐
Contextopdrachten	☐	☐	☐	☐	☐
Projectopdrachten	☐	☐	☐	☐	☐
Interactieve oefeningen	☐	☐	☐	☐	☐

Hoe geschikt vindt u de volgende types oefeningen voor het aanleren van: **Datastructuren**?

(Focus op representatie van data en efficiënt gebruik van structuren, vb lijsten, mappen, bomen,...) *

1 helemaal niet geschikt -> 5= zeer geschikt

	1	2	3	4	5
Gesloten opdrachten	☐	☐	☐	☐	☐
Opdrachten die stapsgewijs opbouwen naar een groter geheel	☐	☐	☐	☐	☐
Contextopdrachten	☐	☐	☐	☐	☐
Projectopdrachten	☐	☐	☐	☐	☐
Interactieve oefeningen	☐	☐	☐	☐	☐

1. Teacher Survey

1 helemaal niet geschikt -> 5= zeer geschikt

	1	2	3	4	5
Gesloten opdrachten	☐	☐	☐	☐	☐
Opdrachten die stapsgewijs opbouwen naar een groter geheel	☐	☐	☐	☐	☐
Contextopdrachten	☐	☐	☐	☐	☐
Projectopdrachten	☐	☐	☐	☐	☐
Interactieve oefeningen	☐	☐	☐	☐	☐

. EXPLORATORY STUDY INSTRUMENTS

Contextualisering van het vak Informaticawetenschappen/equivalent

In het laatste deel van de vragenlijst richten we ons op **contextualisering en interdisciplinair onderwijs**, met een focus op koppelingen tussen programmeren en biologie.

(**Contextualisering** betekent dat programmeerconcepten worden aangeleerd aan de hand van herkenbare situaties of vakinhouden uit andere vakken, zoals biologie)

Welke ondersteuning zou u het meeste helpen bij het geven van gecontextualiseerde informaticalessen binnen BCSW? *

1 = helemaal niet → 5 = zeer helpend

	1	2	3	4	5
Kant-en-klare oefeningen	☐	☐	☐	☐	☐
Voorbeeldlessen	☐	☐	☐	☐	☐
Handleiding voor leraren	☐	☐	☐	☐	☐
Oefeningen met differentiatie	☐	☐	☐	☐	☐

Welke voorwaarden moeten volgens u vervuld zijn om interdisciplinaire programmeeroefeningen met biologie-concepten succesvol in te voeren binnen uw school? *

(Succesvol = realiseren van leerplandoelen én verhogen van betrokkenheid en motivatie.)

(Open antwoord)

.1. Teacher Survey

Einde

Hartelijk dank voor het invullen van deze vragenlijst! Uw input is van grote waarde voor dit onderzoek.

Indien u interesse heeft om in een volgende fase **mee te werken aan het evalueren van programmeeroefeningen** of aan het **testen van (onderdelen van) een curriculum**, neem dan contact op via: gino.vandeemput@student.kuleuven.be

Leerkrachten die deelnemen aan deze volgende fase **krijgen toegang tot het bijhorende lesmateriaal**.

Vergeet niet op **Verzenden** te klikken om uw antwoorden in te dienen.

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.



.2 Semi-Structured Interview Guide

School waar je les geeft? :

Graad waar je les geeft? :

A. Achtergrond & Leerlingen

1.1 Wat is uw achtergrond en huidige vakkenpakket?

- Welke vakken geeft u?
- Wat is uw opleiding?

1.2 Heeft u een formele opleiding in informatica of informaticadidactiek gevolgd?

- Zo nee: hoe heeft u zichzelf voorbereid op het geven van dit vak? (zelfstudie, materiaal collega's, trainingen, YouTube, etc.)

1.3 Hoeveel ervaring heeft u met het geven van informaticawetenschappen / programmeren?

- Hoeveel jaar?
- In welke graden of richtingen?

B. Huidige manier van lesgeven & Oefeningen(*gefocust op informatica*)

2.1 Hoe wordt informatica momenteel op uw school gegeven?

- Apart vak of geïntegreerd met andere vakken?
- Wordt er aan co-teaching gedaan?

2. Hoe zou u uw manier van lesgeven omschrijven?

- Volgt u bepaalde pedagogische principes of methodieken?
- Hoe structureert u doorgaans een les?
- Welke aanpak werkt volgens u het beste?

3. Hoe kiest u de oefeningen of opdrachten die u gebruikt, en welke soorten werken het beste voor uw leerlingen?

- Kleine, afgebakende oefeningen of opdrachten die stapsgewijs uitbreiden?
- Projecten, contextopdrachten, real-life voorbeelden?
- Hoe beoordeeld/evalueert u oefeningen
- geef u verschillende oefeningen afhankelijk van de student?

3.2 heb je de indruk dat studenten de oefeningen leuk vinden, dat ze gemotiveerd zijn

4. Wat vinden leerlingen moeilijk aan informatica / programmeren?

5. Welke digitale middelen gebruiken u en uw leerlingen tijdens de lessen, en hoe goed werkt dit in de praktijk?

- Laptop, tablet?
- Zijn er beperkingen of praktische drempels?
- Verschillen tussen klasgroepen?

6. Gebruikt u zelf tools zoals ChatGPT of andere LLM's bij de voorbereiding van lessen?

- Waarvoor gebruikt u ze?
- Welke voordelen ervaart u?
- Heeft u ook bedenkingen of grenzen?
- Merkt u dat leerlingen LLM's gebruiken, en hoe beïnvloedt dat de lessen?

6.1 Wat mogen leerlingen gebruiken tijdens het maken van oefeningen

-google

-LLM zoals chatgpt

7. Hoe ziet de tijdlijn of volgorde eruit waarin u de verschillende onderdelen van informaticawetenschappen behandelt?

- Welke topics komen eerst, en waarom?
- Verschilt de planning per richting of klasgroep?
- Hoe flexibel bent u in deze volgorde (strikt curriculum vs. ruimte om te schuiven)?
- Zijn er onderdelen die u graag anders zou willen inplannen, en waarom?

D. Contextualisering & Interdisciplinair Werken

8. In welke mate gebruikt u context, real-life toepassingen of voorbeelden in uw lessen?

- Wat werkt goed?
- Wat werkt minder goed?
- Hoe reageren leerlingen op contextopdrachten?

9. Hoe kijkt u aan tegen het contextualiseren van programmeeroefeningen met eenvoudige concepten uit biologie?

- Bijvoorbeeld: “vergelijk 2 dna sequenties -> vind de langste substring, vergelijk base”
- Zou dit motivatie kunnen verhogen?
- Wat zijn mogelijke drempels of aandachtspunten?

9.2 Welke voorwaarden moeten volgens u vervuld zijn om interdisciplinaire ICT-biologie-chemie oefeningen succesvol te gebruiken?

(open antwoord)

10. Werkt u soms samen met leerkrachten van andere vakken of hoe ziet de samenwerking eruit?

- Op welke manieren gebeurt dit eventueel?
 - Zou u openstaan voor meer samenwerking?
 - Welke drempels ziet u (tijd, afstemming, praktische organisatie)?
-

E. Noden & Ondersteuning

11. Wat zijn volgens u de grootste uitdagingen in het huidige curriculum of bij het lesgeven in uw vak(informatica wetenschappen)?

- Conceptuele moeilijkheden (loops, variabelen, abstractie)?
- Motivatieproblemen?
- Niveauverschillen?
- Infrastructuur of tijdsdruk?

12. Wat zou u het meest helpen om uw lessen te versterken?

- Kant-en-klare oefeningen?
 - Interdisciplinaire voorbeelden?
 - Differentiatie?
 - Tools, begeleiding of materiaal?
 - Samenwerking met collega's?
-

6. Slot

13. Is er nog iets dat u belangrijk vindt om mee te geven dat we nog niet besproken hebben?

- Ideeën, zorgen, kansen?
- Toekomstvisie op het vak?

ppendix B

Classroom Study Questionnaire Instruments

B.1 Pre

B. CLASSROOM STUDY QUESTIONNAIRE INSTRUMENTS

Onderzoek motivatie informaticawetenschappen

Bedankt om deel te nemen aan ons onderzoek over je motivatie voor informaticawetenschappen! Hieronder vind je enkele vragen, gelieve deze te beantwoorden. Als er iets niet duidelijk is, mag je dit steeds laten weten.

* Required

B.1. Pre

1. Lees elke uitspraak zorgvuldig en duid aan in welke mate deze jouw reden weerspiegelt om deel te nemen aan het vak Informaticawetenschappen. Gebruik de onderstaande schaal:

- 1 = Helemaal niet
 2 = Zeer weinig
 3 = Weinig
 4 = Matig
 5 = Voldoende
 6 = Veel
 7 = Helemaal

Waarom ben je op dit moment gemotiveerd voor het vak Informaticawetenschappen? *

	1	2	3	4	5	6	7
Omdat ik dit vak interessant vind	☐	☐	☐	☐	☐	☐	☐
Omdat ik het volg voor mijn eigen ontwikkeling	☐	☐	☐	☐	☐	☐	☐
Omdat het verplicht is in mijn studierichting	☐	☐	☐	☐	☐	☐	☐
Er zijn misschien goede redenen om dit vak te volgen, maar persoonlijk zie ik die niet	☐	☐	☐	☐	☐	☐	☐
Omdat ik dit vak leuk vind	☐	☐	☐	☐	☐	☐	☐
Omdat ik denk dat dit vak goed is voor mijn toekomst	☐	☐	☐	☐	☐	☐	☐
Omdat het iets is wat ik moet doen	☐	☐	☐	☐	☐	☐	☐
Ik volg dit vak, maar ik weet niet zeker of het de moeite waard is	☐	☐	☐	☐	☐	☐	☐
Omdat ik plezier beleef aan dit vak	☐	☐	☐	☐	☐	☐	☐
Omdat ik er zelf voor gekozen heb	☐	☐	☐	☐	☐	☐	☐
Omdat ik geen andere keuze heb	☐	☐	☐	☐	☐	☐	☐
Ik weet het niet; ik zie niet wat dit vak mij oplevert	☐	☐	☐	☐	☐	☐	☐
Omdat ik me goed voel tijdens dit vak	☐	☐	☐	☐	☐	☐	☐
Omdat ik geloof dat dit vak belangrijk is voor mij	☐	☐	☐	☐	☐	☐	☐
Omdat ik het gevoel heb dat ik het moet volgen	☐	☐	☐	☐	☐	☐	☐
Ik volg dit vak, maar ik twijfel of het zinvol is om ermee door te gaan	☐	☐	☐	☐	☐	☐	☐

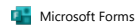
B. CLASSROOM STUDY QUESTIONNAIRE INSTRUMENTS

Uniek ID

2. Uniek ID *

Gelieve dit niet te veranderen en dien de form in!

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.



B.2 Post

B. CLASSROOM STUDY QUESTIONNAIRE INSTRUMENTS

Post bevraging Onderzoek motivatie informaticawetenschappen

* Required

Motivatie meting na interventie

Bedankt om deel te nemen aan ons onderzoek over je motivatie voor informaticawetenschappen! Hieronder vind je enkele vragen, gelieve deze te beantwoorden.

Als er iets niet duidelijk is, mag je dit steeds laten weten.

Wanneer u dit formulier indient, worden uw gegevens, zoals naam en e-mailadres, niet automatisch verzameld, tenzij u het zelf opgeeft.

B.2. Post

1. Lees elke uitspraak zorgvuldig en duid aan in welke mate deze jouw reden weerspiegelt om deel te nemen aan het vak Informaticawetenschappen.

Waarom ben je op dit moment gemotiveerd voor het vak informaticawetenschappen? *

	1 = Helemaal niet	2 = Zeer weinig	3 = Weinig	4 = Matig	5 = Voldoende	6 = Veel	7 = Helemaal
Omdat ik het vak interessant vind	☐	☐	☐	☐	☐	☐	☐
Omdat ik het volg voor mijn eigen ontwikkeling	☐	☐	☐	☐	☐	☐	☐
Omdat het verplicht is in mijn studierichting	☐	☐	☐	☐	☐	☐	☐
Er zijn misschien goede redenen om dit vak te volgen, maar persoonlijk zie ik die niet	☐	☐	☐	☐	☐	☐	☐
Omdat ik dit vak leuk vind	☐	☐	☐	☐	☐	☐	☐
Omdat ik denk dat dit vak goed is voor mijn toekomst	☐	☐	☐	☐	☐	☐	☐
Omdat het iets is wat ik moet doen	☐	☐	☐	☐	☐	☐	☐
Ik volg dit vak, maar ik weet niet zeker of het de moeite waard is	☐	☐	☐	☐	☐	☐	☐
Omdat ik plezier beleef aan dit vak	☐	☐	☐	☐	☐	☐	☐
Omdat ik er zelf voor gekozen heb	☐	☐	☐	☐	☐	☐	☐
Omdat ik geen andere keuze heb	☐	☐	☐	☐	☐	☐	☐
Ik weet het niet; ik zie niet wat dit vak mij oplevert	☐	☐	☐	☐	☐	☐	☐
Omdat ik me goed voel tijdens dit vak	☐	☐	☐	☐	☐	☐	☐
Omdat ik geloof dat dit vak belangrijk is voor mij	☐	☐	☐	☐	☐	☐	☐
Omdat ik het gevoel heb dat ik het moet volgen	☐	☐	☐	☐	☐	☐	☐
Ik volg dit vak, maar ik twijfel of het zinvol is om ermee door te gaan	☐	☐	☐	☐	☐	☐	☐

B. CLASSROOM STUDY QUESTIONNAIRE INSTRUMENTS

Verduidelijking en achterliggende informatie

Kies voor elk van de volgende vragen één/meer van de gegeven oefeningen. Je kan eventueel de oefeningen terug openzetten in een apart tabblad. Als je nog niet alle oefeningen maakte, kies je uit de oefeningen die je al wel maakte.

2. Welke oefeningen vond je fijn/leuk om te maken? *

- ☐ 1.4
- ☐ 2.4
- ☐ 3.4
- ☐ 4.7
- ☐ Geen

3. Welke oefeningen vond je niet fijn/leuk om te maken? *

- ☐ 1.4
- ☐ 2.4
- ☐ 3.4
- ☐ 4.7
- ☐ Geen

4. Welke oefeningen vond je gemakkelijk? *

- ☐ 1.4
- ☐ 2.4
- ☐ 3.4
- ☐ 4.7
- ☐ Geen

5. Welke oefeningen vond je moeilijk? *

- ☐ 1.4
- ☐ 2.4
- ☐ 3.4
- ☐ 4.7
- ☐ Geen

B.2. Post

Verduidelijking en achterliggende informatie

We willen graag weten welke oefeningen jij oploste en hoe ver je geraakte.

6. Loste je oefeningen op met een biologische context? *

☐ Ja

☐ Nee

7. Geef het nummer van de laatste oefening die je maakte. *

B. CLASSROOM STUDY QUESTIONNAIRE INSTRUMENTS

Verduidelijking en achterliggende informatie

In dit laatste deel volgen nog enkele score vragen, geef aan in welke mate je het eens bent met de volgende stellingen.

8. Beantwoord de volgende stelling *

	Veel minder goed	Minder goed	Iets minder goed	Geen verschil	Iets beter	Beter	Veel beter
Geef aan of er een verschil was tussen de voorbije drie lessen en de lessen daarvoor. (Een les kan verschillen op vlak van beleefd plezier, niveau,...)	☐	☐	☐	☐	☐	☐	☐

9. Beantwoord de volgende stelling *

	Helemaal niets bijgeleerd	Zeer weinig bijgeleerd	Weinig bijgeleerd	Matig bijgeleerd	Veel bijgeleerd	Zeer veel bijgeleerd	Heel veel bijgeleerd
Leerde je veel bij over informatica	☐	☐	☐	☐	☐	☐	☐

10. In welke mate ben je het eens met de volgende stellingen? * B.2. Post

	Helemaal oneens	Oneens	Neutraal	Eens	Helemaal eens
De biologische context maakte de programmeeroefeningen interessanter dan gelijkaardige oefeningen zonder context.	☐	☐	☐	☐	☐
De biologische context maakt het nut van de programmeeroefeningen duidelijker.	☐	☐	☐	☐	☐
Ik zou deze programmeeroefeningen liever zonder biologische context maken.	☐	☐	☐	☐	☐
Het niveau van het programmeren was haalbaar.	☐	☐	☐	☐	☐
Ik moest tijdens de oefening extra nadenken over de biologische context om te begrijpen wat er gevraagd werd.	☐	☐	☐	☐	☐
De combinatie van biologie en programmeren maakte de oefening onnodig ingewikkeld.	☐	☐	☐	☐	☐
De biologische context hielp mij om beter te begrijpen wat het programma moest doen.	☐	☐	☐	☐	☐
Tijdens deze oefening voelde ik me mentaal overbelast of gefrustreerd.	☐	☐	☐	☐	☐
De biologische context was gemakkelijk te begrijpen.	☐	☐	☐	☐	☐

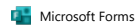
B. CLASSROOM STUDY QUESTIONNAIRE INSTRUMENTS

Uniek ID

11. Uniek ID *

Gelieve dit niet te veranderen en dien de form in!

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.



ppendix C

Teacher and Domain-Expert Evaluation Instruments

C.1 Teacher Questionnaire

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Evaluatie van oefenreeksen met gecontextualiseerde programmeeroefeningen

Bedankt om deel te nemen aan deze studie in het kader van mijn masterproef aan KU Leuven.

Deze vragenlijst peilt naar de bruikbaarheid van automatisch gegenereerde programmeeroefeningen binnen een biologische context. U krijgt drie korte oefenreeksen te zien. Voor elke reeks geldt het volgende:

- De reeks bestaat uit vijf oefeningen.
- De reeks is bedoeld als een eerste stap binnen een bepaald programmeerconcept.
- De reeks behandelt dus niet het volledige programmeerconcept.
- Bij elke reeks wordt vermeld wat het doel van de reeks is.
- Er wordt aangegeven welke programmeerconcepten als voorkennis worden verondersteld.
- Er wordt ook aangegeven welke subtopics binnen de scope vallen en welke onderdelen bewust buiten beschouwing blijven.

Bij elke oefening worden ook beschrijvende testcases getoond. Deze geven aan wat er getest wordt en welke feedback een leerling zou krijgen bij fouten.

Let daarbij vooral op of de feedback duidelijk en bruikbaar is voor leerlingen die hun fouten proberen te begrijpen.

Beoordeel elke reeks in functie van het afgebakende doel, de vermelde voorkennis en de opgegeven scope.

* Required

Achtergrond Informatie

1. Welke vakken geeft u ? *

2. Hoeveel jaar geeft u les? *

Eerste reeks oefeningen variabelen

Neem bij het beantwoorden van deze vragen, reeks "Variabelen.pdf" erbij.

Beoordeel de 5 oefeningen als één geheel. Baseer je antwoord op je algemene indruk van de reeks, niet op één losse oefening. Wanneer één oefening duidelijk zwakker is dan de rest, kan je dat verderop apart aanduiden en kort toelichten.

C.1. Teacher Questionnaire

3. Duidelijkheid en bruikbaarheid *

Focus bij dit onderdeel:

- Beoordeel of de oefeningen duidelijk zijn voor leerlingen
- Beoordeel of de biologische context het programmeren ondersteunt of net bemoeilijkt.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De probleem stellingen van de oefeningen geeft duidelijke instructies aan de leerlingen.	☐	☐	☐	☐	☐
De probleem stelling van de oefeningen geeft teveel informatie in een keer	☐	☐	☐	☐	☐
De biologie context leidt de leerling te veel af van het eigenlijke programmeer oel	☐	☐	☐	☐	☐
Uit de probleemstellin gen is voldoende duidelijk welke programmeerco ncepten leerlingen nodig hebben om de oefeningen op te lossen.	☐	☐	☐	☐	☐

4. Ondersteuning bij het leren *

Focus bij dit onderdeel:

- Beoordeel of de aangeboden ondersteuning leerlingen helpt om tot een oplossing te komen, zonder de oplossing volledig weg te geven.
- Let bij de testcases vooral op of de feedback leerlingen helpt om hun fouten beter te begrijpen.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De templates bieden een gepast niveau van ondersteuning.	☐	☐	☐	☐	☐
De templates bieden genoeg ondersteuning zonder te veel van het antwoord weg te geven.	☐	☐	☐	☐	☐
De modeloplossing en zijn duidelijk en bruikbaar vanuit didactisch oogpunt.	☐	☐	☐	☐	☐
De beschreven test cases helpt leerlingen om hun fouten beter te begrijpen.	☐	☐	☐	☐	☐

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

5. Kwaliteit van de reeks *

Focus bij dit onderdeel:

Beoordeel of de reeks geschikt is als eerste kennismaking met het programmeerconcept. Let daarbij op:

- de opbouw
- de moeilijkheidsgraad
- het aantal nieuwe elementen
- de bruikbaarheid in de klas.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De reeks sluit goed aan bij het vooropgestelde leerdoel.	☐	☐	☐	☐	☐
De reeks is geschikt voor leerlingen die dit programmeerconcept voor het eerst leren.	☐	☐	☐	☐	☐
De reeks is logisch opgebouwd, met een geleidelijke en gepaste toename in moeilijkheid.	☐	☐	☐	☐	☐
Er worden te veel nieuwe elementen tegelijk geïntroduceerd.	☐	☐	☐	☐	☐
De reeks helpt leerlingen om het concept stap voor stap te begrijpen.	☐	☐	☐	☐	☐
Ik zou dit soort materiaal gebruiken in mijn eigen lessen	☐	☐	☐	☐	☐

6. Hoe uitgebreid zouden de aanpassingen moeten zijn voordat u deze reeks in de klas zou gebruiken? *

- ☐ Geen aanpassingen nodig
- ☐ Kleine aanpassingen nodig
- ☐ Matige aanpassingen nodig
- ☐ Grote aanpassingen nodig
- ☐ Ik zou deze reeks niet gebruiken

7. Licht uw antwoord op de vorige vraag kort toe. *

8. Wat is volgens u de belangrijkste aanpassing die nodig is om deze reeks bruikbaar te maken in de klas? *

9. Was er een oefening die qua bruikbaarheid duidelijk zwakker was dan de andere? Zo ja, welke en waarom? *

C.1. Teacher Questionnaire

Tweede reeks oefeningen conditionals

Neem bij het beantwoorden van deze vragen, reeks "Conditionals.pdf" erbij.

Beoordeel de 5 oefeningen als één geheel. Baseer je antwoord op je algemene indruk van de reeks, niet op één losse oefening. Wanneer één oefening duidelijk zwakker is dan de rest, kan je dat verderop apart aanduiden en kort toelichten.

10. Duidelijkheid en bruikbaarheid *

Focus bij dit onderdeel:

- Beoordeel of de oefeningen duidelijk zijn voor leerlingen
- Beoordeel of de biologische context het programmeren ondersteunt of net bemoeilijkt.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De probleem stellingen van de oefeningen geeft duidelijke instructies aan de leerlingen.	☐	☐	☐	☐	☐
De probleem stelling van de oefeningen geeft teveel informatie in een keer	☐	☐	☐	☐	☐
De biologie context leidt de leerling te veel af van het eigenlijke programmeer oel	☐	☐	☐	☐	☐

11. Ondersteuning bij het leren *

Focus bij dit onderdeel:

- Beoordeel of de aangeboden ondersteuning leerlingen helpt om tot een oplossing te komen, zonder de oplossing volledig weg te geven.
- Let bij de testcases vooral op of de feedback leerlingen helpt om hun fouten beter te begrijpen.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De templates bieden een gepast niveau van ondersteuning.	☐	☐	☐	☐	☐
De templates bieden genoeg ondersteuning zonder te veel van het antwoord weg te geven.	☐	☐	☐	☐	☐
De modeloplossing en zijn duidelijk en bruikbaar vanuit didactisch oogpunt.	☐	☐	☐	☐	☐
De beschreven test cases helpt leerlingen om hun fouten beter te begrijpen.	☐	☐	☐	☐	☐

C₁₂ TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Focus bij dit onderdeel:

Beoordeel of de reeks geschikt is als eerste kennismaking met het programmeerconcept. Let daarbij op:

- de opbouw
- de moeilijkheidsgraad
- het aantal nieuwe elementen
- de bruikbaarheid in de klas.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De reeks sluit goed aan bij het vooropgestelde leerdoel.	☐	☐	☐	☐	☐
De reeks is geschikt voor leerlingen die dit programmeerconcept voor het eerst leren.	☐	☐	☐	☐	☐
De reeks is logisch opgebouwd, met een geleidelijke en gepaste toename in moeilijkheid.	☐	☐	☐	☐	☐
Er worden te veel nieuwe elementen tegelijk geïntroduceerd.	☐	☐	☐	☐	☐
De reeks helpt leerlingen om het concept stap voor stap te begrijpen.	☐	☐	☐	☐	☐
Ik zou dit soort materiaal gebruiken in mijn eigen lessen	☐	☐	☐	☐	☐

13. Hoe uitgebreid zouden de aanpassingen moeten zijn voordat u deze reeks in de klas zou gebruiken? *

- ☐ Geen aanpassingen nodig
- ☐ Kleine aanpassingen nodig
- ☐ Matige aanpassingen nodig
- ☐ Grote aanpassingen nodig
- ☐ Ik zou deze reeks niet gebruiken

14. Licht uw antwoord op de vorige vraag kort toe. *

15. Wat is volgens u de belangrijkste aanpassing die nodig is om deze reeks bruikbaar te maken in de klas? *

16. Was er een oefening die qua bruikbaarheid duidelijk zwakker was dan de andere? Zo ja, welke en waarom? *

C.1. Teacher Questionnaire

Derde reeks oefeningen While loops

Neem bij het beantwoorden van deze vragen, reeks "WhileLoops.pdf" erbij.

Beoordeel de 5 oefeningen als één geheel. Baseer je antwoord op je algemene indruk van de reeks, niet op één losse oefening. Wanneer één oefening duidelijk zwakker is dan de rest, kan je dat verderop apart aanduiden en kort toelichten.

17. Duidelijkheid en bruikbaarheid *

Focus bij dit onderdeel:

- Beoordeel of de oefeningen duidelijk zijn voor leerlingen
- Beoordeel of de biologische context het programmeren ondersteunt of net bemoeilijkt.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De probleem stellingen van de oefeningen geeft duidelijke instructies aan de leerlingen.	☐	☐	☐	☐	☐
De probleem stelling van de oefeningen geeft teveel informatie in een keer	☐	☐	☐	☐	☐
De biologie context leidt de leerling te veel af van het eigenlijke programmeer doel	☐	☐	☐	☐	☐

18. Ondersteuning bij het leren *

Focus bij dit onderdeel:

- Beoordeel of de aangeboden ondersteuning leerlingen helpt om tot een oplossing te komen, zonder de oplossing volledig weg te geven.
- Let bij de testcases vooral op of de feedback leerlingen helpt om hun fouten beter te begrijpen.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De templates bieden een gepast niveau van ondersteuning.	☐	☐	☐	☐	☐
De templates bieden genoeg ondersteuning zonder te veel van het antwoord weg te geven.	☐	☐	☐	☐	☐
De modeloplossing en zijn duidelijk en bruikbaar vanuit didactisch oogpunt.	☐	☐	☐	☐	☐
De beschreven test cases helpt leerlingen om hun fouten beter te begrijpen.	☐	☐	☐	☐	☐

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

19. Kwaliteit van de reeks *

Focus bij dit onderdeel:

Beoordeel of de reeks geschikt is als eerste kennismaking met het programmeerconcept. Let daarbij op:

- de opbouw
- de moeilijkheidsgraad
- het aantal nieuwe elementen
- de bruikbaarheid in de klas.

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
De reeks sluit goed aan bij het vooropgestelde leerdoel.	☐	☐	☐	☐	☐
De reeks is geschikt voor leerlingen die dit programmeerconcept voor het eerst leren.	☐	☐	☐	☐	☐
De reeks is logisch opgebouwd, met een geleidelijke en gepaste toename in moeilijkheid.	☐	☐	☐	☐	☐
Er worden te veel nieuwe elementen tegelijk geïntroduceerd.	☐	☐	☐	☐	☐
De reeks helpt leerlingen om het concept stap voor stap te begrijpen.	☐	☐	☐	☐	☐
Ik zou dit soort materiaal gebruiken in mijn eigen lessen	☐	☐	☐	☐	☐

20. Hoe uitgebreid zouden de aanpassingen moeten zijn voordat u deze reeks in de klas zou gebruiken? *

- ☐ Geen aanpassingen nodig
- ☐ Kleine aanpassingen nodig
- ☐ Matige aanpassingen nodig
- ☐ Grote aanpassingen nodig
- ☐ Ik zou deze reeks niet gebruiken

21. Licht uw antwoord op de vorige vraag kort toe. *

22. Wat is volgens u de belangrijkste aanpassing die nodig is om deze reeks bruikbaar te maken in de klas? *

23. Was er een oefening die qua bruikbaarheid duidelijk zwakker was dan de andere? Zo ja, welke en waarom? *

C.1. Teacher Questionnaire

Vragen na alle 3 reeksen

24. Welke van de 3 reeksen vond u het meest geschikt als introductie op een nieuw programmeerconcept, en waarom? *

Reeks 1: variabelen | Reeks 2: conditionals | Reeks 3: while loops

25. Welk terugkerend probleem zag u in meerdere reeksen? *

26. Ziet u zichzelf dit soort materiaal gebruiken in uw eigen lessen? *

	Helemaal niet akkoord	Eerder niet akkoord	Neutraal	Eerder akkoord	Helemaal akkoord
Ziet u zichzelf dit soort materiaal gebruiken in uw eigen lessen?	☐	☐	☐	☐	☐

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Follow-upgesprek

Bent u bereid om deel te nemen aan een aanvullend interview over de geëvalueerde oefeningenreeksen?

Tijdens dit interview wordt verder ingegaan op de bruikbaarheid van de reeksen, mogelijke verbeterpunten en verwachtingen voor toekomstige iteraties van het systeem.

27. Zou u hieraan willen deelnemen? *

☐ Ja

☐ Nee

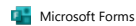
C.1. Teacher Questionnaire

Ik neem graag deel aan het follow-upgesprek

Bedankt dat je wil deelnemen aan het follow-upgesprek. Ik zal verder contact opnemen met u via email om een moment in te plannen.

28. Wat is uw e-mailadres? *

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.



C.2 Domain-Expert Questionnaire

C.2. Domain-Expert Questionnaire

Evaluation of exercise sequences with contextualized programming exercises

Thank you for participating in this study as part of my master's thesis at KU Leuven.

This questionnaire evaluates the usability and quality of automatically generated programming exercises in a biological context.

- Each sequence consists of five exercises.
- Each sequence is intended as a first introduction to a specific programming concept.
- The sequence therefore does not cover the full programming concept.
- For each sequence, the goal of the sequence is stated.
- The assumed prior programming knowledge is also indicated.
- The subtopics that are in scope and the elements that are deliberately out of scope are also specified.

Each exercise also includes descriptive test cases. These indicate what is being tested and what kind of student solutions the test cases are intended to detect. When reviewing these test cases, please pay particular attention to whether they cover the most important incorrect or incomplete solutions for the exercises. Evaluate each sequence in relation to its stated goal, the assumed prior knowledge, and the specified scope.

* Required

Background information

1. Area of expertise? *

2. Years of experience? *

3. Experience with teaching or evaluating beginner programming? *

- ☐ No experience
- ☐ Limited experience
- ☐ Moderate experience
- ☐ Extensive experience

Exercise sequence 1: Variables

Please use the file "Variables.pdf" when answering the following questions.

Evaluate the 5 exercises as one introductory sequence for a new programming concept. Base your answers on your overall impression of the sequence as a whole, rather than on any single exercise. Consider content correctness, technical quality, and the way the sequence introduces the concept step by step. If one exercise is clearly weaker than the others, you can indicate this later and briefly explain why.

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

4. Problem statement and task quality *

Focus on:

- whether the exercises are understandable, precise, and complete enough for students to know what they need to do.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The problem statements are sufficiently clear and precise.	☐	☐	☐	☐	☐
The exercises provide enough information to solve the tasks correctly.	☐	☐	☐	☐	☐
The tasks avoid unnecessary ambiguity or multiple valid interpretations.	☐	☐	☐	☐	☐
The examples, where provided, are relevant and support understanding of the task.	☐	☐	☐	☐	☐
The problem statements make it sufficiently clear which programming concepts students need to solve the exercises.	☐	☐	☐	☐	☐

5. Technical correctness *

Focus on:

- whether the model solutions, templates, and described test cases are technically correct and aligned with the intended scope.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The model solutions correctly implement what is asked in the problem statements.	☐	☐	☐	☐	☐
The model solutions stay within the intended scope of the exercise.	☐	☐	☐	☐	☐
The described test cases cover the most important incorrect or incomplete solutions for the exercises.	☐	☐	☐	☐	☐
The templates provide a technically correct starting structure.	☐	☐	☐	☐	☐

C.2. Domain-Expert Questionnaire

6. Suitability as an introduction *

Focus on:

- whether the sequence works as a first introduction to the target programming concept, without introducing too much at once.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The sequence is appropriate as a first introduction to the target programming concept.	☐	☐	☐	☐	☐
The sequence introduces the concept in a gradual and manageable way.	☐	☐	☐	☐	☐
The exercises do not introduce too many new elements at once.	☐	☐	☐	☐	☐
The sequence remains focused on the intended programming concept without unnecessary detours.	☐	☐	☐	☐	☐

7. Sequence design *

Focus on the progression of the sequence as a whole:

- whether the exercises build on each other
- increase gradually in difficulty,
- contribute to the stated end goal.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
Each exercise contributes meaningfully to the overall learning goal of the sequence.	☐	☐	☐	☐	☐
The overall level of difficulty is appropriate for an introductory sequence.	☐	☐	☐	☐	☐
Earlier exercises prepare the learner sufficiently for the later ones.	☐	☐	☐	☐	☐
The sequence reached its stated end goal.	☐	☐	☐	☐	☐

8. How extensive would the adjustments need to be before you would consider this sequence suitable for use? *

- ☐ No adjustments needed
- ☐ Minor adjustments needed
- ☐ Moderate adjustments needed
- ☐ Major adjustments needed
- ☐ I would not consider this sequence suitable for use

9. Please briefly explain your answer to the previous question. *

117

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

10. What is the main technical or design issue in this sequence? *

11. Was there one exercise that was clearly weaker than the others? If so, which one, and why? *

C.2. Domain-Expert Questionnaire

Exercise sequence 2: Conditionals

Please use the file "Conditionals.pdf" when answering the following questions.

Evaluate the 5 exercises as one introductory sequence for a new programming concept. Base your answers on your overall impression of the sequence as a whole, rather than on any single exercise. Consider content correctness, technical quality, and the way the sequence introduces the concept step by step. If one exercise is clearly weaker than the others, you can indicate this later and briefly explain why.

12. Problem statement and task quality *

Focus on:

- whether the exercises are understandable, precise, and complete enough for students to know what they need to do.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The problem statements are sufficiently clear and precise.	☐	☐	☐	☐	☐
The exercises provide enough information to solve the tasks correctly.	☐	☐	☐	☐	☐
The tasks avoid unnecessary ambiguity or multiple valid interpretations.	☐	☐	☐	☐	☐
The examples, where provided, are relevant and support understanding of the task.	☐	☐	☐	☐	☐
The problem statements make it sufficiently clear which programming concepts students need to solve the exercises.	☐	☐	☐	☐	☐

13. Technical correctness *

Focus on:

- whether the model solutions, templates, and described test cases are technically correct and aligned with the intended scope.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The model solutions correctly implement what is asked in the problem statements.	☐	☐	☐	☐	☐
The model solutions stay within the intended scope of the exercise.	☐	☐	☐	☐	☐
The described test cases cover the most important incorrect or incomplete solutions for the exercises.	☐	☐	☐	☐	☐
The templates provide a technically correct starting structure.	☐	☐	☐	☐	☐

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

14. Suitability as an introduction

Focus on:

- whether the sequence works as a first introduction to the target programming concept, without introducing too much at once.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The sequence is appropriate as a first introduction to the target programming concept.	☐	☐	☐	☐	☐
The sequence introduces the concept in a gradual and manageable way.	☐	☐	☐	☐	☐
The exercises do not introduce too many new elements at once.	☐	☐	☐	☐	☐
The sequence remains focused on the intended programming concept without unnecessary detours.	☐	☐	☐	☐	☐

15. Sequence design *

Focus on the progression of the sequence as a whole:

- whether the exercises build on each other
- increase gradually in difficulty,
- contribute to the stated end goal.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
Each exercise contributes meaningfully to the overall learning goal of the sequence.	☐	☐	☐	☐	☐
The overall level of difficulty is appropriate for an introductory sequence.	☐	☐	☐	☐	☐
Earlier exercises prepare the learner sufficiently for the later ones.	☐	☐	☐	☐	☐
The sequence reached its stated end goal.	☐	☐	☐	☐	☐

16. How extensive would the adjustments need to be before you would consider this sequence suitable for use? *

- ☐ No adjustments needed
- ☐ Minor adjustments needed
- ☐ Moderate adjustments needed
- ☐ Major adjustments needed
- ☐ I would not consider this sequence suitable for use

17. Please briefly explain your answer to the previous question. *

120

C.2. Domain-Expert Questionnaire

18. What is the main technical or design issue in this sequence? *

19. Was there one exercise that was clearly weaker than the others? If so, which one, and why? *

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Exercise sequence 3: While loops

Please use the file "Whileloops.pdf" when answering the following questions.
Evaluate the 5 exercises as one introductory sequence for a new programming concept. Base your answers on your overall impression of the sequence as a whole, rather than on any single exercise. Consider content correctness, technical quality, and the way the sequence introduces the concept step by step. If one exercise is clearly weaker than the others, you can indicate this later and briefly explain why.

20. Problem statement and task quality *

Focus on :

- whether the exercises are understandable, precise, and complete enough for students to know what they need to do.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The problem statements are sufficiently clear and precise.	☐	☐	☐	☐	☐
The exercises provide enough information to solve the tasks correctly.	☐	☐	☐	☐	☐
The tasks avoid unnecessary ambiguity or multiple valid interpretations.	☐	☐	☐	☐	☐
The examples, where provided, are relevant and support understanding of the task.	☐	☐	☐	☐	☐
The problem statements make it sufficiently clear which programming concepts students need to solve the exercises.	☐	☐	☐	☐	☐

21. Technical correctness *

Focus on:

- whether the model solutions, templates, and described test cases are technically correct and aligned with the intended scope.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The model solutions correctly implement what is asked in the problem statements.	☐	☐	☐	☐	☐
The model solutions stay within the intended scope of the exercise.	☐	☐	☐	☐	☐
The described test cases cover the most important incorrect or incomplete solutions for the exercises.	☐	☐	☐	☐	☐
The templates provide a technically correct starting structure.	☐	☐	☐	☐	☐

22. Suitability as an introduction *

C.2. Domain-Expert Questionnaire

Focus on:

- whether the sequence works as a first introduction to the target programming concept, without introducing too much at once.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The sequence is appropriate as a first introduction to the target programming concept.	☐	☐	☐	☐	☐
The sequence introduces the concept in a gradual and manageable way.	☐	☐	☐	☐	☐
The exercises do not introduce too many new elements at once.	☐	☐	☐	☐	☐
The sequence remains focused on the intended programming concept without unnecessary detours.	☐	☐	☐	☐	☐

23. Sequence design *

Focus on the progression of the sequence as a whole:

- whether the exercises build on each other
- increase gradually in difficulty,
- contribute to the stated end goal.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
Each exercise contributes meaningfully to the overall learning goal of the sequence.	☐	☐	☐	☐	☐
The overall level of difficulty is appropriate for an introductory sequence.	☐	☐	☐	☐	☐
Earlier exercises prepare the learner sufficiently for the later ones.	☐	☐	☐	☐	☐
The sequence reached its stated end goal.	☐	☐	☐	☐	☐

24. How extensive would the adjustments need to be before you would consider this sequence suitable for use? *

- ☐ No adjustments needed
- ☐ Minor adjustments needed
- ☐ Moderate adjustments needed
- ☐ Major adjustments needed
- ☐ I would not consider this sequence suitable for use

25. Please briefly explain your answer to the previous question. *

123

C. ~~TEACHER AND DOMAIN-EXPERT~~ EVALUATION INSTRUMENTS

~~26. What is the main technical or design issue in this sequence? *~~

27. Was there one exercise that was clearly weaker than the others? If so, which one, and why? *

C.2. Domain-Expert Questionnaire

Questions after all three sequences

28. Which of the 3 sequences was the strongest as an introduction to a new programming concept, and why? *

Sequence 1: variables | Sequence 2: conditionals | Sequence 3: while loops

29. What recurring weakness did you notice across multiple sequences? *

30. What is the single most important improvement you would suggest for these introductory sequences? *

31. Would you consider this type of material suitable for educational use? *

Strongly disagree

Disagree

Neutral

Agree

Strongly agree

Would you
consider this
type of material
suitable for
educational
use?

☐☐☐☐☐

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Follow-up interview

Are you willing to take part in a follow-up interview about the evaluated exercise sequences?

During this interview, we will further discuss the quality of the sequences, possible improvements, and expectations for future iterations of the system.

32. Would you be willing to participate? *

☐ Yes

☐ No

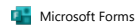
C.2. Domain-Expert Questionnaire

I would like to participate in the follow-up interview

Thank you for agreeing to participate in the follow-up interview. I will contact you by email to schedule a suitable time.

33. What is your email address? *

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.



C.3 Variable exercise sequence

C.3.1 English

Generated Exercises

Introduction to the exercise sequence

This exercise sequence introduces **variables** using a biological context around **DN bases and DN structure**. The exercises use the nucleobases `A`, `T`, `C`, and `G` as concrete examples for storing values in variables.

The goal of the sequence is for students to learn how to create variables, assign values, overwrite values, and use meaningful variable names. The focus is on representing simple biological data, such as a base or a base pair.

Assumed prior knowledge

Students are assumed to already be familiar with:

- general computer literacy.

Topics within scope

- variable assignment;
- literals such as strings and integers;
- re-assignment;
- using variables in simple expressions;
- naming variables.

The sequence builds from storing one nucleobase to using multiple variables to represent a biological relationship, such as base pairing. By the end, students use variables to record both a nucleobase symbol and its position.

Topics outside the scope

- user input;
 - type casting;
 - complex data structures such as lists;
 - Boolean logic;
 - loops;
 - complex string operations such as slicing.
-

Exercise 1: Storing a Nucleobase in a Variable

Concepts

- Variable naming
- String literal
- Assignment operator (`=`)
- `print()`

Other Subject Concepts

- Biology: DN nucleobases (`A` = adenine, `T` = Thymine, `C` = Cytosine, `G` = Guanine)

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Problem Description

DNA carries hereditary information and consists of four different building blocks, the nucleobases: adenine (A), Thymine (T), Cytosine (C), and Guanine (G). To work with this information in a computer program, we need to store it in memory using variables.

In this exercise, you will store the letter A in a variable. Choose a meaningful name for your variable, for example `nucleobase`, so that someone reading your code immediately understands what it refers to. Then use the `print()` function to display the value of your variable on the screen.

Expected output:

Scaffolded Template

```
# TODO: Create a variable with a meaningful name (e.g. nucleobase) and assign it the value 'A'

# TODO: Use the print() function to display the content of your variable on the screen
```

Solution Code

```
nucleobase = 'A'
print(nucleobase)
```

Test Cases

```
## Test case

### Goal of the test case
```

This test case checks whether the student first stores the DNA base 'A' in a variable and then prints it.

The expected output is one line with exactly 'A'.

In addition, the test case checks whether the student does not use a lowercase letter and does not print more than necessary.

```
### What is tested?
```

- Whether the student prints something using `print()`.
- Whether the student prints exactly the uppercase letter 'A'.
- Whether the student does not accidentally print the lowercase letter 'a'.
- Whether the student prints only one line.
- Whether the student first stores the value 'A' in a variable.
- Whether the student does not only use `print('A')` directly without a variable.

```
### Feedback the student may receive
```

- `Traceback (most recent call last):`
- `File "solution.py", line 1, in <module>`
- `print('a')`
- `TypeError: 'a' is not a valid ASCII character`
- `You did not use the print() function to display the value of the variable.`
- `Note: DNA bases are always written in uppercase letters. You printed 'a' instead of 'A'.`
- `You printed '{actual_output}', but the expected output is 'A'.`
- `Your code prints too many lines. Use print() only to display the value of the variable.`
- `You did not store the value 'A' in a variable. The assignment asks you to create a variable first.`

Exercise 2: DN Base Pairs and Meaningful Variables

Concepts

- Multiple variable names
- Assignment operator (=)
- String literals
- print()

Other Subject Concepts

- Biology (DN structure)
- Complementary base pairing (C pairs with G)

Problem Description

In the previous exercise, you learned how to store one nucleobase, such as `C`, in a variable. In biology, however, these bases form fixed pairs. One important pair in DN is Cytosine (C) and Guanine (G). They always belong together.

In this exercise, we will link the base `C` to its partner `G`. To do this, you will use two different variables. It is important to give your variables names that clearly indicate what is stored in them.

Assignment:

1. Look at the variable `first_base` in the code below.
2. Create a second variable and store the letter `G` in it. Give this variable a meaningful name, for example `partner_base`.
3. Use the `print()` function to display first the first variable and then the second variable on the screen.

Expected output:

```
C
G
```

Scaffolded Template

```
# Step 1: The first base has already been stored in a variable
first_base = 'C'

# TODO: Create a second variable for the partner base ('G')
# Use a meaningful name for your variable

# TODO: Use the print() function to display both variables underneath each other
```

Solution Code

```
first_base = 'C'
partner_base = 'G'
print(first_base)
print(partner_base)
```

Test Cases

```
## Test case

### Goal of the test case
```

This test case checks whether the student keeps the existing variable `first_base` with the value `'C'`.

In addition, the student must create a second variable with the value `'G'`.

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

The test case also checks whether the two values are printed underneath each other in the correct order: first ``C``, then ``G``.

What is tested?

- Whether the student has not removed the variable ``first_base``.
- Whether the student keeps the correct value ``C`` in the variable ``first_base``.
- Whether the student creates a second variable with the value ``G``.
- Whether the student uses the uppercase letter ``G`` instead of the lowercase letter ``g``.
- Whether the student prints exactly two lines.
- Whether the student prints first ``C`` and then ``G``.
- Whether the student uses a meaningful name for the second variable.

Feedback the student may receive

- ``The variable first_base was not found. Did you accidentally remove it?``
- ``The variable first_base must contain the value C, not {first_base}.``
- ``You did not create a second variable with the value G.``
- ``Python is case-sensitive. Use an uppercase G for the nucleobase.``
- `` n error occurred while printing: {e}``
- ``We expected at least 2 lines of output (C and G), but received {len(output_lines)}.``
- ``You printed too many lines ({len(output_lines)}). Print only the two variables underneath each other.``
- ``The first line must be C, but it says: {output_lines[0]}.``
- ``The second line must be G, but it says: {output_lines[1]}.``
- ``The name of your second variable ({partner_var_name}) is very short. Use a meaningful name such as partner_base.``

Exercise 3: Point Mutations: Changing a Base

Concepts

- Assignment operator (`=`)
- Strings (text)
- `print()` function
- Overwriting variables (re-assignment)

Other Subject Concepts

- Biology: DN base pairs
- Biology: point mutations

Problem Description

In the previous exercise, you learned how to store a base pair, such as Cytosine (C) and Guanine (G), in two different variables. In biology, however, something can change in the DN . This is called a mutation. In a point mutation, one base is replaced by another.

In this exercise, we will simulate a point mutation in Python. You start with a variable `base` that has the value (`denine`). Your task is to overwrite this value with the letter `G` (Guanine).

Hint: In the previous exercise, you used `=` to give a variable a value. You can use the same symbol to give an existing variable a new value. The old value is then erased and replaced.

Assignment:

1. The code already contains a variable `base` with the value `denine` and prints it.
2. Overwrite the variable `base` with the new letter `G`.
3. Print the variable `base` again so that you can see the difference.

Expected output:

G

Scaffolded Template

```
base = ' '
print(base)

# TODO: Overwrite the variable 'base' with the new value 'G'

# TODO: Print the new value of the variable 'base' on the screen
```

Solution Code

```
base = ' '
print(base)
base = 'G'
print(base)
```

Test Cases

Test case

Goal of the test case

This test case checks whether the student keeps the existing variable `base` and overwrites its value to `G`.

The test case mainly looks at the variable itself, not at the printed output.

The expected situation is that `base` eventually contains the value `G` and is still a string.

What is tested?

- Whether the student uses the existing variable `base`.
- Whether the student changes the value of `base` to `G`.
- Whether the student overwrites the variable `base` instead of using a new variable.
- Whether the student keeps `base` as a text value.
- Whether the student uses the uppercase letter `G` instead of the lowercase letter `g`.

Feedback the student may receive

- `The variable 'base' was not found. Make sure you use the existing variable.`
- `The variable 'base' currently has the value '{base}', but it should be 'G'. Did you overwrite the value with base = 'G'?`
- `The variable 'base' must remain a string (text), for example 'G'.`
- `Python is case-sensitive. You used 'g' instead of the uppercase letter 'G'.`

Exercise 4: DN Base Pairs and Clear Variable Names

Concepts

- String literal
- Meaningful variable names
- Assignment operator (=)
- print()

Other Subject Concepts

- Biology (DNA base pairing - T and C-G)

Problem Description

In the previous exercise, you learned how to change the value stored in a variable (re-assignment). In double-stranded DNA, however, bases always occur in pairs: adenine (A) pairs with Thymine (T), and Cytosine (C) pairs with Guanine (G).

To keep the code clear, it is important to choose names that describe exactly what is stored in the variable. Instead of a general name such as `base`, we now use `left_base` and `right_base`. This immediately makes it clear that the variables refer to a pair.

Assignment:

1. The variable `left_base` currently has the value `'A'`. Overwrite this value with `G` (a mutation).
2. Create a new variable named `right_base`. Give it the value that pairs with `G` according to the rules of base pairing (C).
3. Use the `print()` function to display first `left_base` and then `right_base`.

Expected output:

```
G
C
```

Scaffolded Template

```
# We start with the left base 'A'
left_base = 'A'

# STEP 1: Change the value of left_base to 'G'
# Hint: Use the assignment operator (=) as in the previous exercise
# TODO: Overwrite left_base with 'G'

# STEP 2: Create a new variable for the right side of the pair
# The complementary base of 'G' is 'C'
# TODO: Create the variable 'right_base' and give it the value 'C'

# STEP 3: Print both variables underneath each other
# TODO: Print left_base
# TODO: Print right_base
```

Solution Code

```
left_base = 'A'
left_base = 'G'
right_base = 'C'
print(left_base)
print(right_base)
```

Test Cases

```
## Test case
```

```
### Goal of the test case
```

This test case checks whether the student uses two variables for a base pair: ``left_base`` and ``right_base``.

The value of ``left_base`` must be overwritten to ``G``.

The value of ``right_base`` must be ``C``, the complement of ``G``.

What is tested?

- Whether the student has defined the variable ``left_base``.
- Whether the student has defined the variable ``right_base``.
- Whether the student correctly overwrites ``left_base`` to ``G``.
- Whether the student correctly sets ``right_base`` to ``C``.
- Whether the student understands that ``C`` is the complement of ``G``.
- Whether the student prints exactly two lines.
- Whether the student first prints ``G`` and then ``C``.

Feedback the student may receive

- ``The variable left_base is not defined.``
- ``The variable right_base is not defined.``
- ``The value of left_base should be G, but is now {l_base}. Did you overwrite the original ?``
- ``The value of right_base should be C (the complement of G), but is now {r_base}.``
- ``Something went wrong while printing: {e}``
- ``{len(lines)} lines were printed, but we expected exactly 2.``
- ``The first printed line must be G, not {lines[0]}.``
- ``The second printed line must be C, not {lines[1]}.``
- ``Test failed: {e}``
- ``Unexpected error: {e}``

Exercise 5: Base Pairs and Their Position

Concepts

- String literal
- Integer literal
- Meaningful variable names
- Assignment operator (`=`)
- `print()`

Other Subject Concepts

- DN nucleobases
- Base pairing
- DN sequences

Problem Description

In the previous exercise, you learned how to use meaningful names such as `left_base` and `right_base` to describe a base pair. However, a DN sequence is a long chain of these pairs. To remember the order, we give each pair a sequence number. This is called the position.

In this exercise, we will use an integer literal, a whole number, to store the position. While we always use quotation marks for letters (strings), we do not use quotation marks for numbers (integers).

Assignment:

1. Extend the existing code by creating a new variable `position`.
2. Give this variable the value 1. Use an integer literal.
3. Use three separate `print()` functions to display `left_base`, `right_base`, and `position` in the console, in that order.

Expected output:

G
C
1

Scaffolded Template

```
left_base = 'G'
right_base = 'C'

# TODO: Create a variable named position and give it the value 1

# TODO: Print the value of left_base

# TODO: Print the value of right_base

# TODO: Print the value of position
```

Solution Code

```
left_base = 'G'
right_base = 'C'
position = 1
print(left_base)
print(right_base)
print(position)
```

Test Cases

Test case

Goal of the test case

This test case checks whether the student creates a new variable `position` with the value `1`.

The variable `position` must be a whole number, not a text value such as `"1"`.

In addition, the test case checks whether the existing variables `left_base` and `right_base` keep their values and whether the three values are printed in the correct order.

What is tested?

- Whether the student has created the variable `position`.
- Whether the student uses `position` as an integer and not as text.
- Whether the student gives `position` the value `1`.
- Whether the student keeps the value `G` in `left_base`.
- Whether the student keeps the value `C` in `right_base`.
- Whether the student prints exactly three lines.
- Whether the student first prints `G`, then `C`, and then `1`.

Feedback the student may receive

- `Error: You have not created the variable position yet.`
- `Error: The variable position must be an integer (whole number). Did you perhaps use quotation marks?`
- `Error: The variable position must have the value 1, but we found {position}.`
- `Error: The variable left_base must keep the value G.`

- *`Error: The variable right\_base must keep the value C.`*
- *`Error while printing: one of the variables is not defined ({e}).`*
- *`You must print exactly 3 lines using three separate print functions. We found {len(output)}.`*
- *`The first line must be G, but {output[0]} was printed.`*
- *`The second line must be C, but {output[1]} was printed.`*
- *`The third line must be 1, but {output[2]} was printed.`*

C.3.2 Dutch

Generated Exercises

Introductie van de oefeningenreeks

Deze oefeningenreeks introduceert **variabelen** aan de hand van een biologische context rond **DN -basen en DN -structuur**. De oefeningen gebruiken de nucleobasen , T, C en G als concrete voorbeelden om waarden op te slaan in variabelen.

Het doel van de reeks is dat leerlingen leren om variabelen aan te maken, waarden toe te kennen, waarden te overschrijven en betekenisvolle variabelenamen te gebruiken. De focus ligt op het representeren van eenvoudige biologische gegevens, zoals een base of een basepaar.

Veronderstelde voorkennis

Leerlingen worden verondersteld al vertrouwd te zijn met:

- algemene computergeletterdheid.

Onderwerpen binnen de scope

- variabele toekenning;
- literals zoals strings en integers;
- re-assignment;
- variabelen gebruiken in eenvoudige expressies;
- naamgeving van variabelen.

De reeks bouwt op van het opslaan van één nucleobase naar het gebruiken van meerdere variabelen om een biologische relatie, zoals baseparing, weer te geven. Op het einde gebruiken leerlingen variabelen om zowel een symbool als een positie van een nucleobase vast te leggen.

Onderwerpen buiten de scope

- input van de gebruiker;
- type casting;
- complexe datastructuren zoals lijsten;
- boolean logica;
- lussen;
- complexe string-operaties zoals slicing.

Exercise 1: Een Nucleobase Opslaan in een Variabele

Concepts

- Variabele naamgeving
- String literal
- assignment operator (=)
- print()

Other Subject Concepts

- Biologie: DN -nucleobasen (denine, Thymine, Cytosine, Guanine)

Problem Description

DN is de drager van erfelijke informatie en is opgebouwd uit vier verschillende bouwstenen, de nucleobasen: Adenine (A), Thymine (T), Cytosine (C) en Guanine (G). Om met deze gegevens te werken in een computerprogramma, moeten we ze opslaan in het geheugen met behulp van variabelen.

In deze oefening ga je de letter ' ' opslaan in een variabele. Kies een betekenisvolle naam voor je variabele (bijvoorbeeld 'nucleobase') zodat iemand die je code leest direct begrijpt waar het over gaat. Daarna gebruik je de print-functie om de waarde van je variabele op het scherm te tonen.

Verwachte uitvoer:

Scaffolded Template

```
# TODO: Maak een variabele aan met een betekenisvolle naam (bijv. nucleobase) en ken de waarde ' ' toe

# TODO: Gebruik de print() functie om de inhoud van jouw variabele op het scherm te tonen
```

Solution Code

```
nucleobase = ' '
print(nucleobase)
```

Test Cases

```
## Testcase
```

```
### Doel van de testcase
```

Deze testcase controleert of de leerling de DN -base ' ' eerst opslaat in een variabele en daarna print.

De verwachte uitvoer is één regel met exact ' '.

Daarnaast controleert de testcase of de leerling geen kleine letter gebruikt en niet meer print dan nodig.

```
### Wat wordt getest?
```

- Of de leerling iets print met `print()`.
- Of de leerling exact de hoofdletter ' ' print.
- Of de leerling niet per ongeluk de kleine letter 'a' print.
- Of de leerling slechts één regel print.
- Of de leerling de waarde ' ' eerst opslaat in een variabele.
- Of de leerling niet alleen rechtstreeks `print(' ')` gebruikt zonder variabele.

```
### Feedback die de leerling kan krijgen
```

- ``Er is een fout opgetreden bij het uitvoeren van de code: {e}``
- ``Je hebt de print() functie niet gebruikt om de waarde van de variabele te tonen.``
- ``Let op: DN -basen worden altijd met hoofdletters geschreven. Je printte 'a' in plaats van ' '.``
- ``Je hebt '{actual_output}' geprint, maar de verwachte uitvoer is ' '.``
- ``Je code print te veel regels. Gebruik print() alleen om de waarde van de variabele te tonen.``
- ``Je hebt de waarde ' ' niet opgeslagen in een variabele. De opdracht vraagt om eerst een variabele aan te maken.``

Exercise 2: DN -basenparen en Betekenisvolle Variabelen

Concepts

- Meerdere variabele namen
- Assignment operator (=)
- String literals
- print()

Other Subject Concepts

- Biologie (DN -structuur)
- Complementaire base-paring (C koppelt aan G)

Problem Description

In de vorige oefening heb je geleerd hoe je één nucleobase (zoals ' ') opslaat in een variabele. In de biologie vormen deze basen echter vaste paren. Een belangrijk paar in DN is Cytosine (C) en Guanine (G). Ze horen altijd bij elkaar.

In deze oefening gaan we de base 'C' koppelen aan zijn partner 'G'. Je gaat hiervoor twee verschillende variabelen gebruiken. Het is belangrijk dat je variabelen namen geeft die duidelijk maken wat erin is opgeslagen.

Opdracht: 1. Bekijk de variabele 'eerste_base' in de code hieronder. 2. Maak een tweede variabele aan en sla hierin de letter 'G' op. Geef deze variabele een betekenisvolle naam, bijvoorbeeld 'partner_base'. 3. Gebruik de print()-functie om eerst de eerste variabele en daarna de tweede variabele op het scherm te tonen.

Verwachte uitvoer: C G

Scaffolded Template

```
# Stap 1: De eerste base is al opgeslagen in een variabele
eerste_base = 'C'

# TODO: Maak een tweede variabele aan voor de partner-base ('G')
# Gebruik een betekenisvolle naam voor je variabele

# TODO: Gebruik de print() functie om beide variabelen onder elkaar te tonen
```

Solution Code

```
eerste_base = 'C'
partner_base = 'G'
print(eerste_base)
print(partner_base)
```

Test Cases

```
## Testcase

### Doel van de testcase
```

Deze testcase controleert of de leerling de bestaande variabele ``eerste_base`` behoudt met de waarde ``C``.

Daarnaast moet de leerling een tweede variabele aanmaken met de waarde ``G``.

De testcase controleert ook of de twee waarden onder elkaar worden geprint in de juiste volgorde: eerst ``C``, daarna ``G``.

Wat wordt getest?

- Of de leerling de variabele ``eerste_base`` niet heeft verwijderd.
- Of de leerling de variabele ``eerste_base`` de correcte waarde ``C`` laat behouden.
- Of de leerling een tweede variabele aanmaakt met de waarde ``G``.
- Of de leerling de hoofdletter ``G`` gebruikt in plaats van de kleine letter ``g``.
- Of de leerling precies twee regels print.
- Of de leerling eerst ``C`` print en daarna ``G``.
- Of de leerling een betekenisvolle naam gebruikt voor de tweede variabele.

Feedback die de leerling kan krijgen

- ``De variabele eerste_base is niet gevonden. Heb je deze per ongeluk verwijderd?``
- ``De variabele eerste_base moet de waarde C bevatten, niet {eerste_base}.``
- ``Je hebt geen tweede variabele aangemaakt met de waarde G.``
- ``Python is hoofdlettergevoelig. Gebruik een hoofdletter G voor de nucleobase.``
- ``Er is een fout opgetreden bij het printen: {e}``
- ``We verwachtten minstens 2 regels output (C en G), maar kregen er {len(output_lines)}.``
- ``Je hebt te veel regels geprint ({len(output_lines)}). Print alleen de twee variabelen onder elkaar.``
- ``De eerste regel moet C zijn, maar er staat: {output_lines[0]}.``
- ``De tweede regel moet G zijn, maar er staat: {output_lines[1]}.``
- ``De naam van je tweede variabele ({partner_var_name}) is erg kort. Gebruik een betekenisvolle naam zoals partner_base.``

Exercise 3: Puntmutaties: Een Base Veranderen

Concepts

- Toekenningsoperator (`=`)
- Strings (tekst)
- `print()` functie
- Variabelen overschrijven (re-assignment)

Other Subject Concepts

- Biologie: DN -basenparen
- Biologie: Puntmutaties

Problem Description

In de vorige oefening heb je geleerd hoe je een basenpaar, zoals Cytosine (C) en Guanine (G), opslaat in twee verschillende variabelen. In de biologie kan er echter iets veranderen in het DN . Dit noemen we een mutatie. Bij een puntmutatie wordt één base vervangen door een andere.

In deze oefening gaan we een puntmutatie nabootsen in Python. Je begint met een variabele `'base'` die de waarde `'A'` (adenine) heeft. Jouw taak is om deze waarde te overschrijven met de letter `'G'` (Guanine).

Hint: Je hebt in de vorige oefening de `'='` gebruikt om een variabele een waarde te geven. Je kunt hetzelfde symbool gebruiken om een bestaande variabele een nieuwe waarde te geven. De oude waarde wordt dan gewist en vervangen.

Opdracht: 1. De code bevat al een variabele `'base'` met de waarde `'A'` en print deze. 2. Overschrijf de variabele `'base'` met de nieuwe letter `'G'`. 3. Print de variabele `'base'` opnieuw zodat je het verschil ziet.

Verwachte uitvoer: `A` `G`

Scaffolded Template

```
base = 'A'
print(base)
```

```
# TODO: Overschrijf de variabele 'base' met de nieuwe waarde 'G'

# TODO: Print de nieuwe waarde van de variabele 'base' op het scherm
```

Solution Code

```
base = ' '
print(base)
base = 'G'
print(base)
```

Test Cases

```
## Testcase
```

```
### Doel van de testcase
```

Deze testcase controleert of de leerling de bestaande variabele ``base`` behoudt en de waarde ervan overschrijft naar ``G``.

De testcase kijkt vooral naar de variabele zelf, niet naar de geprinte uitvoer.

De verwachte situatie is dat ``base`` uiteindelijk de waarde ``G`` bevat en nog steeds een string is.

```
### Wat wordt getest?
```

- Of de leerling de bestaande variabele ``base`` gebruikt.
- Of de leerling de waarde van ``base`` aanpast naar ``G``.
- Of de leerling de variabele ``base`` overschrijft in plaats van een nieuwe variabele te gebruiken.
- Of de leerling ``base`` als tekstwaarde bewaart.
- Of de leerling de hoofdletter ``G`` gebruikt in plaats van de kleine letter ``g``.

```
### Feedback die de leerling kan krijgen
```

- ``De variabele 'base' is niet gevonden. Zorg ervoor dat je de bestaande variabele gebruikt.``
- ``De variabele 'base' heeft momenteel de waarde '{base}', maar dit moet 'G' zijn. Heb je de waarde overschreven met base = 'G'?``
- ``De variabele 'base' moet een string (tekst) blijven, bijvoorbeeld 'G'.``
- ``Python is hoofdlettergevoelig. Je hebt 'g' gebruikt in plaats van de hoofdletter 'G'.``

Exercise 4: DN Base-paren en Duidelijke Variabelenamen

Concepts

- String literal
- Betekenisvolle variabelenamen
- Assignment operator (=)
- print()

Other Subject Concepts

- Biologie (DN base-pairing -T en C-G)

Problem Description

In de vorige oefening heb je geleerd hoe je een waarde in een variabele kunt aanpassen (re-assignment). In dubbelstrengs DN komen basen echter altijd in paren voor: denine () koppelt aan Thymine (T), en Cytosine

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

(C) koppelt aan Guanine (G).

Om de code overzichtelijk te houden, is het belangrijk om namen te kiezen die precies vertellen wat er in de variabele zit. In plaats van een algemene naam zoals 'base', gebruiken we nu 'linker_base' en 'rechter_base'. Zo snapt iedereen direct dat het om een paar gaat.

Opdracht: 1. De variabele 'linker_base' heeft nu de waarde ' '. Overschrijf deze waarde met 'G' (een mutatie). 2. Maak een nieuwe variabele aan met de naam 'rechter_base'. Geef deze de waarde die bij 'G' hoort volgens de regels van base-pairing (C). 3. Gebruik de print() functie om eerst de 'linker_base' en daarna de 'rechter_base' te tonen.

Verwachte uitvoer: G C

Scaffolded Template

```
# We beginnen met de linker base ' '
linker_base = ' '

# ST P 1: Pas de waarde van linker_base aan naar 'G'
# Hint: Gebruik de assignment operator (=) zoals in de vorige oefening
# TODO: Overschrijf linker_base met 'G'

# ST P 2: Maak een nieuwe variabele aan voor de rechterkant van het paar
# De complementaire base van 'G' is 'C'
# TODO: Maak de variabele 'rechter_base' aan en geef deze de waarde 'C'

# ST P 3: Print beide variabelen onder elkaar
# TODO: Print linker_base
# TODO: Print rechter_base
```

Solution Code

```
linker_base = ' '
linker_base = 'G'
rechter_base = 'C'
print(linker_base)
print(rechter_base)
```

Test Cases

```
## Testcase
```

```
### Doel van de testcase
```

Deze testcase controleert of de leerling twee variabelen gebruikt voor een basenpaar:

``linker_base`` en ``rechter_base``.

De waarde van ``linker_base`` moet overschreven worden naar ``G``.

De waarde van ``rechter_base`` moet ``C`` zijn, het complement van ``G``.

```
### Wat wordt getest?
```

- Of de leerling de variabele ``linker_base`` heeft gedefinieerd.
- Of de leerling de variabele ``rechter_base`` heeft gedefinieerd.
- Of de leerling ``linker_base`` correct overschrijft naar ``G``.
- Of de leerling ``rechter_base`` correct instelt op ``C``.
- Of de leerling begrijpt dat ``C`` het complement is van ``G``.
- Of de leerling precies twee regels print.
- Of de leerling eerst ``G`` print en daarna ``C``.

```
### Feedback die de leerling kan krijgen
```

```
- `De variabele linker_base is niet gedefinieerd.`
- `De variabele rechter_base is niet gedefinieerd.`
- `De waarde van linker_base zou G moeten zijn, maar is nu {l_base}. Heb je de oorspronkelijke
  wel overschreven?`
- `De waarde van rechter_base zou C moeten zijn (het complement van G), maar is nu {r_base}.`
- `Er ging iets mis bij het printen: {e}`
- `Er werden {len(lines)} regels geprint, maar we verwachtten er precies 2.`
- `De eerste geprinte regel moet G zijn, niet {lines[0]}.`
- `De tweede geprinte regel moet C zijn, niet {lines[1]}.`
- `Test mislukt: {e}`
- `Onverwachte fout: {e}`
```

Exercise 5: Base-paren en hun Positie

Concepts

- String literal
- Integer literal
- Betekenisvolle variabelenamen
- Assignment operator (=)
- print()

Other Subject Concepts

- DN -nucleobasen
- Base-pairing
- DN -sequenties

Problem Description

In de vorige oefening heb je geleerd hoe je betekenisvolle namen zoals ‘linker_base’ en ‘rechter_base’ gebruikt om een base-paar te beschrijven. Een DN -sequentie is echter een lange keten van deze paren. Om de volgorde te onthouden, geven we elk paar een volgnummer. Dit noemen we de positie.

In deze oefening gaan we een ‘Integer literal’ (een geheel getal) gebruiken om de positie op te slaan. Terwijl we voor letters altijd aanhalingstekens gebruiken (Strings), doen we dat bij getallen (Integers) niet.

Opdracht: 1. Breid de bestaande code uit door een nieuwe variabele ‘positie’ aan te maken. 2. Geef deze variabele de waarde 1 (gebruik een integer literal). 3. Gebruik drie aparte print() functies om achtereenvolgens de ‘linker_base’, de ‘rechter_base’ en de ‘positie’ in de console te tonen.

Verwachte uitvoer: G C 1

Scaffolded Template

```
linker_base = 'G'
rechter_base = 'C'
```

```
# TODO: Maak een variabele genaamd positie aan en geef deze de waarde 1
```

```
# TODO: Print de waarde van linker_base
```

```
# TODO: Print de waarde van rechter_base
```

```
# TODO: Print de waarde van positie
```

Solution Code

```
linker_base = 'G'
rechter_base = 'C'
positie = 1
print(linker_base)
print(rechter_base)
print(positie)
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling een nieuwe variabele ``positie`` aanmaakt met de waarde ``1``.

De variabele ``positie`` moet een geheel getal zijn, dus geen tekstwaarde zoals ``"1"``.

Daarnaast controleert de testcase of de bestaande variabelen ``linker_base`` en ``rechter_base`` hun waarden behouden en of de drie waarden in de juiste volgorde worden geprint.

Wat wordt getest?

- Of de leerling de variabele ``positie`` heeft aangemaakt.
- Of de leerling ``positie`` als integer gebruikt en niet als tekst.
- Of de leerling ``positie`` de waarde ``1`` geeft.
- Of de leerling ``linker_base`` de waarde ``G`` laat behouden.
- Of de leerling ``rechter_base`` de waarde ``C`` laat behouden.
- Of de leerling precies drie regels print.
- Of de leerling eerst ``G``, daarna ``C``, en daarna ``1`` print.

Feedback die de leerling kan krijgen

- ``Fout: Je hebt de variabele positie nog niet aangemaakt.``
- ``Fout: De variabele positie moet een integer (geheel getal) zijn. Heb je misschien aanhalingstekens gebruikt?``
- ``Fout: De variabele positie moet de waarde 1 hebben, maar we vonden {positie}.``
- ``Fout: De variabele linker_base moet de waarde G behouden.``
- ``Fout: De variabele rechter_base moet de waarde C behouden.``
- ``Fout bij het printen: een van de variabelen is niet gedefinieerd ({e}).``
- ``Je moet precies 3 regels printen met drie aparte print-functies. We vonden er {len(output)}.``
- ``De eerste regel moet G zijn, maar er werd {output[0]} geprint.``
- ``De tweede regel moet C zijn, maar er werd {output[1]} geprint.``
- ``De derde regel moet 1 zijn, maar er werd {output[2]} geprint.``

C.4 Conditionals exercise sequence

Generated Exercises

Introduction to the exercise sequence

This exercise sequence introduces **conditional statements** using a biological context around **gene expression, codons, and proteins**. The exercises use start and stop codons as concrete examples for making decisions in code.

The goal of the sequence is for students to learn how to use **if**, **else**, and **elif** to classify biological data. The focus is on recognizing different types of codons, such as start codons, stop codons, and other coding codons.

Assumed prior knowledge

Students are assumed to already be familiar with:

- variables;
- simple string values.

Topics within scope

- **if**;
- **if/else**;
- **elif**.

The sequence builds from one simple check with **if** to multiple branches with **if/elif/else**. By the end, students classify a codon as **Start**, **Stop**, or **Coding**.

Topics outside the scope

- nested conditional statements;
 - compound conditions with **and** or **or**;
 - **<**, **>**.
-

Exercise 1: Recognizing the start codon

Concepts

- Variables
- Strings
- If structure
- Comparison with **==**

Other Subject Concepts

- Biology: DNA, codons, and the start codon

Problem Description

In biology, DNA is read in groups of three letters, which are called codons. One specific codon, **TG**, is the start codon: it indicates where translation into a protein should begin. In this exercise, you learn how to use an if structure to check whether a text value (string) matches a specific value.

Task: Complete the code so that the program checks whether the variable **codon** is exactly equal to the text **TG**. If this is the case, the program should display the sentence **This is a start codon!**.

Example: - Input: codon = TG -> Output: This is a start codon! - Input: codon = TGG -> Output: (no output)

Scaffolded Template

```
# The variable codon contains a group of three bases
codon = ' TG'

# Use an if structure to check whether the codon is a start codon
# TODO: Write an if statement that checks whether codon is equal to ' TG'
# TODO: Use print() to display the text 'This is a start codon!' if the condition is true
```

Solution Code

```
codon = ' TG'

if codon == ' TG':
    print('This is a start codon!')
```

Test Cases

```
## Test case
```

```
### Goal of the test case
```

This test case checks whether the student uses an `if` structure to recognize the start codon `TG`.

If `codon` is exactly equal to `TG`, the sentence `This is a start codon!` should be printed.

In all other cases, nothing should be printed.

```
### What is tested?
```

- Whether the student does not cause an error when the code is executed.
- Whether the student prints the correct message when `codon` is equal to `TG`.
- Whether the student prints nothing when `codon` is equal to `TGG`.
- Whether the student compares case-sensitively and does not recognize `atg` as a start codon.
- Whether the student compares exactly with `TG`.
- Whether the student does not use the `in` operator, which would incorrectly recognize values such as `TG`.
- Whether the student prints at most one line.

```
### Feedback the student can receive
```

- `Internal error: student_code was not found.`
- `The code caused an error during execution: {e}`
- `For codon = ' TG', we expected exactly the sentence '{expected_msg}', but got: '{output_atg}'`
- `Error: If the codon is 'TGG', nothing should be printed. Your code printed: '{output_tgg}'. Check your if structure.`
- `Note: DN codes are case-sensitive. The value 'atg' (lowercase letters) should not be recognized as a start codon.`
- `Your code also prints something for ' TG'. The comparison must be exact (use == and not 'in').`
- `You print too many lines ({len(output_lines)}). Only one line may appear on the screen.`

Exercise 2: Recognizing stop codons with if/else

Concepts

- Variables
- Strings
- Comparison with ==
- if/else structure
- Text output with print

Other Subject Concepts

- Biology (DNA translation)
- Stop codons (TGA)

Problem Description

In the previous exercise, you learned how to use an if structure to check whether a codon is the start codon TGA. But what if you also want to do something when the condition is *not* true? For that, we use an else clause.

In biology, translation (the translation of genetic code into a protein) stops at a stop codon, such as TGA. In this exercise, you write a program that makes a binary decision: is the codon a stop codon or not?

Task: Complete the code so that the program checks whether the variable `codon` is equal to TGA. - If the condition is true, print exactly: **This is a stop codon!** - In all other cases (**else**), print exactly: **This is not a stop codon.**

Example 1: - Input: `codon = TGA` - Output: This is a stop codon!

Example 2: - Input: `codon = CCC` - Output: This is not a stop codon.

Scaffolded Template

```
codon = 'TGA' # You can change this to '   ' to test the other branch of the if structure

# Use an if/else structure to check the codon
if codon == 'TGA':
    # TODO: Use print to indicate that it is a stop codon

else:
    # TODO: Use print to indicate that it is NOT a stop codon
```

Solution Code

```
codon = 'TGA'

if codon == 'TGA':
    print('This is a stop codon!')
else:
    print('This is not a stop codon.')
```

Test Cases

```
## Test case
```

```
### Goal of the test case
```

This test case checks whether the student uses an `if-else` structure to recognize the stop codon `TGA`.

If `codon` is exactly equal to `TGA`, the sentence `This is a stop codon!` should be printed.

For other codons, such as `` ``, ``tag``, or ``GGG``, the sentence ``This is not a stop codon.`` should be printed.

What is tested?

- Whether the student does not cause an error when the code is executed.
- Whether the student prints the correct message when ``codon`` is equal to ``T G``.
- Whether the student correctly uses the ``else`` clause for a codon that is not a stop codon.
- Whether the student compares case-sensitively and does not recognize ``tag`` as ``T G``.
- Whether the student prints the expected text exactly, including the exclamation mark in ``This is a stop codon!``.
- Whether the student also executes the ``else`` clause for another random codon, such as ``GGG``.

Feedback the student can receive

- `` n error occurred while executing the code: {e}``
 - ``Error: For codon = 'T G', we expected exactly the text '{expected_tag}', but got: '{result_tag}'``
 - ``Error: For codon = ' ', we expected exactly the text '{expected_aaa}', but got: '{result_aaa}'``
 - ``Error: Python is case-sensitive. 'tag' (lowercase letters) should not be recognized as the stop codon 'T G'`.``
 - ``Error: You probably forgot the exclamation mark (!) at the end of the sentence.``
 - ``Error: For a random codon such as 'GGG', the else clause should be executed.``
-

Exercise 3: Three scenarios: Recognizing multiple stop codons

Concepts

- codon variable
- comparison with `==`
- `elif` clause
- `else` clause
- text output with `print`

Other Subject Concepts

- Biology (DNA translation, stop codons)

Problem Description

In the previous exercise, you learned how to use an `if/else` structure to check whether a codon is the stop codon `T G`. However, in biology there are multiple stop codons that mark the end of translation. To create a program that can choose between three different situations, we use the `elif` clause.

With `if`, you check the first condition. With `elif` (short for “else if”), you add an extra specific check for when the first condition is not true. The `else` is only executed if none of the previous conditions are true.

Task: Complete the code so that the program checks the variable `codon` for the following three scenarios: 1. If `codon` is equal to `T G`, print exactly: `This is stop codon T G!` 2. If `codon` is equal to `T ``, print exactly: `This is stop codon T `!` 3. In all other cases, print exactly: `This is not a stop codon of this type.`

Example 1: - Input: `codon = T G` - Output: `This is stop codon T G!`

Example 2: - Input: `codon = T `` - Output: `This is stop codon T `!`

Example 3: - Input: `codon = GGC` - Output: `This is not a stop codon of this type.`

Scaffolded Template

```
codon = 'T '

if codon == 'T G':
    # TODO: Print the correct text for scenario 1
    p ss
# TODO: dd an elif clause here for codon 'T '
# TODO: Print the correct text for scenario 2
else:
    # TODO: Print the correct text for scenario 3
    p ss
```

Solution Code

```
codon = 'T '

if codon == 'T G':
    print('This is stop codon T G!')
elif codon == 'T ':
    print('This is stop codon T !')
else:
    print('This is not a stop codon of this type.')
```

Test Cases

Test case

Goal of the test case

This test case checks whether the student uses an `if-elif-else` structure to recognize different stop codons.

If `codon` is equal to `T G`, the student should print `This is stop codon T G!`.

If `codon` is equal to `T` , the student should print `This is stop codon T !`.

For other values, such as `GGC` and `TG` , the student should print `This is not a stop codon of this type.`

What is tested?

- Whether the student does not cause an error when the code is executed.
- Whether the student prints the correct message when `codon` is equal to `T G`.
- Whether the student prints the correct message when `codon` is equal to `T` .
- Whether the student correctly uses the `else` clause for other codons.
- Whether the student does not forget to print something for the different test values.
- Whether the student prints the expected texts exactly.
- Whether the student uses an `if-elif-else` structure as requested in the task.

Feedback the student can receive

- `Something went wrong while executing your code for {desc}: {e}`
- `Your code prints nothing when codon {desc} is. Use the print() function.`
- `Error for {desc}.`

Expected: {expected}

Your output: {actual}

- `Your code works for the test values, but you did not use "elif". Use an if-elif-else structure as requested in the task.`

Exercise 4: Four scenarios: Recognizing all three stop codons

Concepts

- Variables
- Strings
- Comparison with ==
- Multiple elif clauses
- Else clause
- Output with print

Other Subject Concepts

- Biology (DNA and translation)
- Stop codons (TGA, TAA, TAG)

Problem Description

In the previous exercise, you learned how to use an if/elif/else structure to choose between three situations. However, in biology there are three different stop codons that mark the end of translation: TGA, TAA, and TAG. To show a separate message for each specific stop codon, we need a structure with two elif clauses.

Remember: we use if for the first check. If that is not true, the computer looks at the elif blocks that follow. Only if nothing matches is the else executed.

Task: Complete the code so that the program checks the variable `codon` for the following four scenarios: 1. If `codon` is equal to TGA, print exactly: `This is stop codon TGA!` 2. If `codon` is equal to TAA, print exactly: `This is stop codon TAA!` 3. If `codon` is equal to TAG, print exactly: `This is stop codon TAG!` 4. In all other cases, print exactly: `This is not a known stop codon.`

Example 1: - Input: `codon = TAG` - Output: `This is stop codon TAG!`

Example 2: - Input: `codon = GGC` - Output: `This is not a known stop codon.`

Scaffolded Template

```
codon = 'TG '
```

```
if codon == 'TGA':
    print('This is stop codon TGA!')
# TODO: dd an elif clause that checks whether codon is equal to 'TAA'

# TODO: dd another elif clause that checks whether codon is equal to 'TAG'

else:
    print('This is not a known stop codon.')
```

Solution Code

```
codon = 'TG '
```

```
if codon == 'TGA':
    print('This is stop codon TGA!')
elif codon == 'TAA':
    print('This is stop codon TAA!')
elif codon == 'TAG':
    print('This is stop codon TAG!')
else:
    print('This is not a known stop codon.')
```

Test Cases

`## Test case`

`### Goal of the test case`

This test case checks whether the student correctly recognizes multiple stop codons with a selection structure.

The codons ``T G``, ``T ``, and ``TG `` should each produce a specific stop codon message.

For an unknown codon, such as ``GGC``, the student should print the general message ``This is not a known stop codon.``

`### What is tested?`

- Whether the student does not cause an error when the code is executed.
- Whether the student correctly recognizes ``T G`` as a stop codon.
- Whether the student correctly recognizes ``T `` as a stop codon.
- Whether the student correctly recognizes ``TG `` as a stop codon.
- Whether the student prints the correct ``else`` message for an unknown codon.
- Whether the student works case-sensitively and does not recognize ``tag`` as ``T G``.
- Whether the student prints the expected texts exactly.

`### Feedback the student can receive`

- ``The code could not be executed for codon '{codon_input}'. Error message: {e}``
 - ``Error for codon '{codon_input}':`
 Expected: ``{expected_output}``
 Your code printed: ``{actual}``

Exercise 5: Classifying codons: Start, stop, or coding?

Concepts

- Variables
- Strings
- Comparisons (==)
- Multiple elif clauses
- Else clause
- Print output

Other Subject Concepts

- DNA bases
- Codons (triplets)
- Start codon (`TG`)
- Stop codons (`T G`, `T ``, `TG ``)
- Translation

Problem Description

In biology, genetic information is translated into proteins. This process always starts at a specific codon: the **start codon** (`TG`). The process ends when one of the three stop codons (`T G`, `T ``, or `TG ``) is reached. All other combinations of three bases are called **coding codons**.

In the previous exercise, you learned how to distinguish three scenarios using `if`, `elif`, and `else`. Now we expand the structure to five possible branches so that we can name each type of codon specifically.

C.4. Conditionals exercise sequence

Task: Complete the code so that the program checks the variable `codon` for the following five scenarios: 1. If `codon` is equal to `TG`, print exactly: `This is the start codon ( TG)`. 2. If `codon` is equal to `T G`, print exactly: `This is stop codon T G`. 3. If `codon` is equal to `T` , print exactly: `This is stop codon T` . 4. If `codon` is equal to `TG` , print exactly: `This is stop codon TG` . 5. In all other cases, print exactly: `This is a coding codon`.

Example 1: - Input: `codon = TG` - Output: `This is the start codon ( TG)`.

Example 2: - Input: `codon = GCT` - Output: `This is a coding codon`.

Scaffolded Template

```
codon = ' TG'

if codon == ' TG':
    # TODO: Print the message for the start codon

elif codon == 'T G':
    # TODO: Print the message for stop codon T G

elif codon == 'T ':
    # TODO: Print the message for stop codon T 

elif codon == 'TG ':
    # TODO: Print the message for stop codon TG 

else:
    # TODO: Print the message for all other coding codons
```

Solution Code

```
codon = ' TG'

if codon == ' TG':
    print('This is the start codon ( TG).')
elif codon == 'T G':
    print('This is stop codon T G.')
elif codon == 'T ':
    print('This is stop codon T .')
elif codon == 'TG ':
    print('This is stop codon TG .')
else:
    print('This is a coding codon.')
```

Test Cases

Test case

Goal of the test case

This test case checks whether the student correctly recognizes different types of codons.

The codon ``TG`` should be recognized as a start codon.

The codons ``T G``, ``T ``, and ``TG `` should be recognized as stop codons.

Other codons, such as ``GCT`` and ``TTC``, should be recognized as coding codons.

What is tested?

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

- Whether the student prints something for each tested codon.
- Whether the student correctly recognizes ``TG`` as a start codon.
- Whether the student correctly recognizes ``T G`` as a stop codon.
- Whether the student correctly recognizes ``T `` as a stop codon.
- Whether the student correctly recognizes ``TG `` as a stop codon.
- Whether the student correctly treats other codons as coding codons.
- Whether the student prints the expected text exactly.
- Whether the student pays attention to capitalization, spaces, and punctuation.

Feedback the student can receive

```
- `Error: Nothing was printed for the situation '{label}'.`  
- `Error while testing {label}.  
Expected: '{expected}'  
Your output: '{actual_output}'  
Tip: Pay attention to capitalization, spaces, and the period at the end.`  
- ` n unexpected error occurred while testing '{codon_val}': {e}`
```

C.4.1 Dutch

Generated Exercises

Introductie van de oefeningenreeks

Deze oefeningenreeks introduceert **conditionele instructies** aan de hand van een biologische context rond **genexpressie, codons en eiwitten**. De oefeningen gebruiken start- en stopcodons als concrete voorbeelden om beslissingen in code te maken.

Het doel van de reeks is dat leerlingen leren om **if**, **else** en **elif** te gebruiken om biologische gegevens te classificeren. De focus ligt op het herkennen van verschillende soorten codons, zoals startcodons, stopcodons en andere codons.

Veronderstelde voorkennis

Leerlingen worden verondersteld al vertrouwd te zijn met:

- variabelen;
- eenvoudige stringwaarden;

Onderwerpen binnen de scope

- **if**;
- **if/else**;
- **elif**.

De reeks bouwt op van één eenvoudige controle met **if** naar meerdere vertakkingen met **if/elif/else**. Op het einde classificeren leerlingen een codon als **Start**, **Stop** of **Coding**.

Onderwerpen buiten de scope

- geneste conditionele instructies;
 - samengestelde voorwaarden met **and** of **or**.
 - **<**, **>**
-

Exercise 1: Herkenning van het startcodon

Concepts

- Variabelen
- Strings
- If-structuur
- Vergelijking met **==**

Other Subject Concepts

- Biologie: DN , codons en het startcodon

Problem Description

In de biologie wordt DN gelezen in groepjes van drie letters, die we codons noemen. Een specifiek codon, ‘ TG’, is het startcodon: het geeft aan waar de vertaling naar een eiwit moet beginnen. In deze oefening leer je hoe je met een if-structuur kunt controleren of een tekst (string) overeenkomt met een specifieke waarde.

Opdracht: Vul de code aan zodat het programma controleert of de variabele ‘codon’ exact gelijk is aan de tekst ‘ TG’. ¹⁵⁸Is dat het geval is, moet het programma de zin ‘Dit is een startcodon!’ op het scherm tonen.

Voorbeeld: - Input: codon = ' TG' -> Output: Dit is een startcodon! - Input: codon = 'TGG' -> Output: (geen output)

Scaffolded Template

```
# De variabele codon bevat een groepje van drie basen
codon = ' TG'

# Gebruik een if-structuur om te controleren of het codon een startcodon is
# TODO: Schrijf een if-statement die controleert of codon gelijk is aan ' TG'
# TODO: Gebruik print() om de tekst 'Dit is een startcodon!' weer te geven als de voorwaarde klopt
```

Solution Code

```
codon = ' TG'

if codon == ' TG':
    print('Dit is een startcodon!')
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling een `if`-structuur gebruikt om het startcodon `TG` te herkennen.

Is `codon` exact gelijk is aan `TG`, moet de zin `Dit is een startcodon!` geprint worden.

In andere gevallen mag er niets geprint worden.

Wat wordt getest?

- Of de leerling geen fout veroorzaakt tijdens het uitvoeren van de code.
- Of de leerling de juiste boodschap print wanneer `codon` gelijk is aan `TG`.
- Of de leerling niets print wanneer `codon` gelijk is aan `TGG`.
- Of de leerling hoofdlettergevoelig vergelijkt en `atg` niet herkent als startcodon.
- Of de leerling exact vergelijkt met `TG`.
- Of de leerling niet de `in`-operator gebruikt waardoor bijvoorbeeld `TG` foutief wordt herkend.
- Of de leerling maximaal één regel print.

Feedback die de leerling kan krijgen

- `Interne fout: student_code niet gevonden.`
- `De code veroorzaakte een fout bij uitvoering: {e}`
- `Bij codon = ' TG' verwachten we exact de zin '{expected_msg}', maar kregen: '{output_atg}'`
- `Fout: ls het codon 'TGG' is, mag er niets geprint worden. Jouw code printte: '{output_tgg}'. Controleer je if-structuur.`
- `Let op: DN -codes zijn hoofdlettergevoelig. De waarde 'atg' (kleine letters) mag niet herkend worden als startcodon.`
- `Je code print ook iets bij ' TG'. De vergelijking moet exact zijn (gebruik == en geen 'in').`
- `Je print te veel regels ({len(output_lines)}). Er mag er maar 1 op het scherm verschijnen.`

Exercise 2: Stopcodons Herkennen met if/else

Concepts

- Variabelen
- Strings
- Vergelijking met ==
- if/else-structuur
- Tekstuele output via print

Other Subject Concepts

- Biologie (DN -translatie)
- Stopcodons (T G)

Problem Description

In de vorige oefening heb je geleerd hoe je met een if-structuur kunt controleren of een codon het startcodon 'T G' is. Maar wat als je ook iets wilt doen wanneer de conditie *niet* waar is? Daarvoor gebruiken we een else-clausule.

In de biologie stopt de translatie (het vertalen van de genetische code naar een eiwit) bij een stopcodon, zoals 'T G'. In deze oefening schrijf je een programma dat een binaire beslissing neemt: is het codon een stopcodon of niet?

Opdracht: Vul de code aan zodat het programma controleert of de variabele 'codon' gelijk is aan 'T G'. - Is de conditie waar is, print dan exact: 'Dit is een stopcodon!' - In alle andere gevallen (else), print dan exact: 'Dit is geen stopcodon.'

Voorbeeld 1: - Input: codon = 'T G' - Output: Dit is een stopcodon!

Voorbeeld 2: - Input: codon = 'CCC' - Output: Dit is geen stopcodon.

Scaffolded Template

```
codon = 'T G' # Je kunt dit veranderen naar '   ' om de andere kant van de if-structuur te
              testen

# Gebruik een if/else-structuur om het codon te controleren
if codon == 'T G':
    # TODO: Gebruik print om aan te geven dat het een stopcodon is

else:
    # TODO: Gebruik print om aan te geven dat het GEEN stopcodon is
```

Solution Code

```
codon = 'T G'

if codon == 'T G':
    print('Dit is een stopcodon!')
else:
    print('Dit is geen stopcodon.')
```

Test Cases

```
## Testcase
```

```
### Doel van de testcase
```

Deze testcase controleert of de leerling een `if-else`-structuur gebruikt om het stopcodon 'T G' te herkennen.

Is ``codon`` exact gelijk is aan ``T G``, moet de zin ``Dit is een stopcodon!`` geprint worden.

Voor andere codons, zoals `` ``, ``tag`` of ``GGG``, moet de zin ``Dit is geen stopcodon.`` geprint worden.

Wat wordt getest?

- Of de leerling geen fout veroorzaakt tijdens het uitvoeren van de code.
- Of de leerling de juiste boodschap print wanneer ``codon`` gelijk is aan ``T G``.
- Of de leerling de ``else``-clausule correct gebruikt voor een codon dat geen stopcodon is.
- Of de leerling hoofdlettergevoelig vergelijkt en ``tag`` niet herkent als ``T G``.
- Of de leerling de verwachte tekst exact print, inclusief het uitroepteken bij ``Dit is een stopcodon!``.
- Of de leerling ook bij een ander willekeurig codon, zoals ``GGG``, de ``else``-clausule uitvoert.

Feedback die de leerling kan krijgen

- ``Er trad een fout op bij het uitvoeren van de code: {e}``
- ``Fout: Bij codon = 'T G' verwachten we exact de tekst '{expected_tag}', maar kregen: '{result_tag}'``
- ``Fout: Bij codon = ' ' verwachten we exact de tekst '{expected_aaa}', maar kregen: '{result_aaa}'``
- ``Fout: Python is hoofdlettergevoelig. 'tag' (kleine letters) mag niet herkend worden als het stopcodon 'T G'`.``
- ``Fout: Je bent waarschijnlijk het uitroepteken (!) vergeten aan het einde van de zin.``
- ``Fout: Voor een willekeurig codon zoals 'GGG' moet de else-clausule uitgevoerd worden.``

Exercise 3: Drie Scenario's: Meerdere Stopcodons Herkennen

Concepts

- variabele codon
- vergelijking met `==`
- `elif`-clausule
- `else`-clausule
- tekstuele output via `print`

Other Subject Concepts

- Biologie (DN -translatie, stopcodons)

Problem Description

In de vorige oefening heb je geleerd hoe je met een `if/else`-structuur kunt controleren of een codon het stopcodon `'T G'` is. In de biologie zijn er echter meerdere stopcodons die het einde van de translatie markeren. Om een programma te maken dat tussen drie verschillende situaties kan kiezen, gebruiken we de `'elif'`-clausule.

Met `'if'` controleer je de eerste conditie. Met `'elif'` (een afkorting voor `'else if'`) voeg je een extra specifieke controle toe voor als de eerste niet waar is. De `'else'` wordt alleen uitgevoerd als geen enkele van de bovenstaande condities waar zijn.

Opdracht: Vul de code aan zodat het programma de variabele `'codon'` controleert op de volgende drie scenario's: 1. Is `'codon'` gelijk is aan `'T G'`, print dan exact: `'Dit is stopcodon T G!'` 2. Is `'codon'` gelijk is aan `'T ' '`, print dan exact: `'Dit is stopcodon T ' !'` 3. In alle andere gevallen, print dan exact: `'Dit is geen stopcodon van dit type.'`

Voorbeeld 1: - Input: `codon = 'T G'` - Output: `Dit is stopcodon T G!`

Voorbeeld 2: - Input: `codon = 'T ' '` - Output: `Dit is stopcodon T ' !`

161

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Voorbeeld 3: - Input: codon = 'GGC' - Output: Dit is geen stopcodon van dit type.

Scaffolded Template

```
codon = 'T  '
```

```
if codon == 'T G':
    # TODO: Print de juiste tekst voor scenario 1
    p ss
# TODO: Voeg hier een elif-clausule toe voor codon 'T  '
# TODO: Print de juiste tekst voor scenario 2
else:
    # TODO: Print de juiste tekst voor scenario 3
    p ss
```

Solution Code

```
codon = 'T  '
```

```
if codon == 'T G':
    print('Dit is stopcodon T G!')
elif codon == 'T  ':
    print('Dit is stopcodon T  !')
else:
    print('Dit is geen stopcodon van dit type.')
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling een ``if-elif-else``-structuur gebruikt om verschillende stopcodons te herkennen.

ls ``codon`` gelijk is aan ``T G``, moet de leerling ``Dit is stopcodon T G!`` printen.

ls ``codon`` gelijk is aan ``T ``, moet de leerling ``Dit is stopcodon T !`` printen.

Voor andere waarden, zoals ``GGC`` en ``TG ``, moet de leerling ``Dit is geen stopcodon van dit type.`` printen.

Wat wordt getest?

- Of de leerling geen fout veroorzaakt tijdens het uitvoeren van de code.
- Of de leerling de juiste boodschap print wanneer ``codon`` gelijk is aan ``T G``.
- Of de leerling de juiste boodschap print wanneer ``codon`` gelijk is aan ``T ``.
- Of de leerling de ``else``-clausule correct gebruikt voor andere codons.
- Of de leerling niets vergeet te printen bij de verschillende testwaarden.
- Of de leerling de verwachte teksten exact print.
- Of de leerling een ``if-elif-else``-structuur gebruikt zoals gevraagd in de opdracht.

Feedback die de leerling kan krijgen

- ``Er ging iets mis bij het uitvoeren van je code voor {desc}: {e}``
 - ``Je code print niets uit wanneer codon {desc} is. Gebruik de print() functie.``
 - ``Fout bij {desc}.`
- Verwacht:162expected}

Jouw output: {actual}`

- *`Je code werkt voor de testwaarden, maar je hebt geen "elif" gebruikt. Gebruik een if-elif-else structuur zoals gevraagd in de opdracht.`*

Exercise 4: Vier Scenario's: Alle Drie de Stopcodons Herkennen

Concepts

- Variabelen
- Strings
- Vergelijking met ==
- Meerdere elif-clausules
- Else-clausule
- Output via print

Other Subject Concepts

- Biologie (DNA en translatie)
- Stopcodons (TGA, TAG, TGA)

Problem Description

In de vorige oefening heb je geleerd hoe je met een if/elif/else-structuur kon kiezen tussen drie situaties. In de biologie zijn er echter drie verschillende stopcodons die het einde van de translatie markeren: TGA, TAG en TGA. Om voor elk specifiek stopcodon een eigen bericht te tonen, hebben we een structuur nodig met twee 'elif'-clausules.

Herinner je: we gebruiken 'if' voor de eerste controle. Als die niet waar is, kijkt de computer naar de 'elif'-blokken die volgen. Pas als helemaal niets klopt, wordt de 'else' uitgevoerd.

Opdracht: Vul de code aan zodat het programma de variabele 'codon' controleert op de volgende vier scenario's: 1. Als 'codon' gelijk is aan 'TGA', print dan exact: 'Dit is stopcodon TGA!' 2. Als 'codon' gelijk is aan 'TAG', print dan exact: 'Dit is stopcodon TAG!' 3. Als 'codon' gelijk is aan 'TGA', print dan exact: 'Dit is stopcodon TGA!' 4. In alle andere gevallen, print dan exact: 'Dit is geen bekend stopcodon.'

Voorbeeld 1: - Input: codon = 'TGA' - Output: Dit is stopcodon TGA !

Voorbeeld 2: - Input: codon = 'GGC' - Output: Dit is geen bekend stopcodon.

Scaffolded Template

```
codon = 'TGA'

if codon == 'TGA':
    print('Dit is stopcodon TGA!')
# TODO: Voeg een elif-clausule toe die controleert of codon gelijk is aan 'TAG'

# TODO: Voeg nog een elif-clausule toe die controleert of codon gelijk is aan 'TGA'

else:
    print('Dit is geen bekend stopcodon.')
```

Solution Code

```
codon = 'TGA'

if codon == 'TGA':
    print('Dit is stopcodon TGA!')
elif codon == 'TAG':
    print('Dit is stopcodon TAG!')
elif codon == 'TGA':
    print('Dit is stopcodon TGA!')
```

```
else:
    print('Dit is geen bekend stopcodon.')
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling meerdere stopcodons correct herkent met een keuzestructuur.

De codons ``T G``, ``T `` en ``TG `` moeten elk een specifieke stopcodonmelding geven.

Voor een onbekend codon, zoals ``GGC``, moet de leerling de algemene melding ``Dit is geen bekend stopcodon.`` printen.

Wat wordt getest?

- Of de leerling geen fout veroorzaakt tijdens het uitvoeren van de code.
- Of de leerling ``T G`` correct herkent als stopcodon.
- Of de leerling ``T `` correct herkent als stopcodon.
- Of de leerling ``TG `` correct herkent als stopcodon.
- Of de leerling voor een onbekend codon de juiste ``else``-melding print.
- Of de leerling hoofdlettergevoelig werkt en ``tag`` niet herkent als ``T G``.
- Of de leerling de verwachte teksten exact print.

Feedback die de leerling kan krijgen

```
- `De code kon niet worden uitgevoerd voor codon '{codon_input}'. Foutmelding: {e}`
- `Fout bij codon '{codon_input}':
Verwacht: '{expected_output}'
Jouw code printte: '{actual}'`
```

Exercise 5: Codons indelen: Start, Stop of Coderend?

Concepts

- Variabelen
- Strings
- Vergelijkingen (==)
- Meerdere elif-clausules
- Else-clausule
- Print-output

Other Subject Concepts

- DN -basen
- Codons (Drietallen)
- Startcodon (`TG`)
- Stopcodons (`T G`, `T ``, `TG ``)
- Translatie

Problem Description

In de biologie wordt genetische informatie vertaald naar eiwitten. Dit proces begint altijd bij een specifiek codon: het **startcodon** (`TG`). Het proces eindigt wanneer er een van de drie stopcodons (`T G`, `T `` of `TG ``) wordt bereikt. **164** Andere combinaties van drie basen noemen we **coderende codons**.

C.4. Conditionals exercise sequence

In de vorige oefening heb je geleerd hoe je drie scenario's kon onderscheiden met 'if', 'elif' en 'else'. Nu gaan we de structuur uitbreiden naar vijf mogelijke takken zodat we elk type codon specifiek kunnen benoemen.

Opdracht: Vul de code aan zodat het programma de variabele 'codon' controleert op de volgende vijf scenario's: 1. ls 'codon' gelijk is aan ' ATG', print exact: 'Dit is het startcodon (ATG)'. 2. ls 'codon' gelijk is aan 'T G', print exact: 'Dit is stopcodon T G'. 3. ls 'codon' gelijk is aan 'T ', print exact: 'Dit is stopcodon T '. 4. ls 'codon' gelijk is aan 'TG ', print exact: 'Dit is stopcodon TG '. 5. In alle andere gevallen, print exact: 'Dit is een coderend codon.'

Voorbeeld 1: - Input: codon = ' ATG' - Output: Dit is het startcodon (ATG).

Voorbeeld 2: - Input: codon = 'GCT' - Output: Dit is een coderend codon.

Scaffolded Template

```
codon = ' ATG'

if codon == ' ATG':
    # TODO: Print de boodschap voor het startcodon

elif codon == 'T G':
    # TODO: Print de boodschap voor stopcodon T G

elif codon == 'T ':
    # TODO: Print de boodschap voor stopcodon T

elif codon == 'TG ':
    # TODO: Print de boodschap voor stopcodon TG

else:
    # TODO: Print de boodschap voor alle overige coderende codons
```

Solution Code

```
codon = ' ATG'

if codon == ' ATG':
    print('Dit is het startcodon ( ATG).')
elif codon == 'T G':
    print('Dit is stopcodon T G.')
elif codon == 'T ':
    print('Dit is stopcodon T .')
elif codon == 'TG ':
    print('Dit is stopcodon TG .')
else:
    print('Dit is een coderend codon.')
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling verschillende types codons correct herkent.

Het codon ` ATG` moet herkend worden als startcodon.

De codons `T G`, `T ` en `TG ` moeten herkend worden als stopcodons.

165

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

ndere codons, zoals ``GCT`` en ``TTC``, moeten herkend worden als coderende codons.

Wat wordt getest?

- Of de leerling iets print voor elk getest codon.
- Of de leerling ``TG`` correct herkent als startcodon.
- Of de leerling ``T G`` correct herkent als stopcodon.
- Of de leerling ``T `` correct herkent als stopcodon.
- Of de leerling ``TG `` correct herkent als stopcodon.
- Of de leerling andere codons correct behandelt als coderende codons.
- Of de leerling de verwachte tekst exact print.
- Of de leerling let op hoofdletters, spaties en leestekens.

Feedback die de leerling kan krijgen

- *`Fout: Er werd niets geprint voor de situatie '{label}'.`*
 - *`Fout bij het testen van {label}.*
- Verwacht: '{expected}'
Jouw output: '{actual_output}'
Tip: Let op hoofdletters, spaties en de punt aan het einde.`
- *`Er trad een onverwachte fout op bij het testen van '{codon\_val}': {e}`*

C.5 While loop exercise sequence

Generated Exercises

Introduction to the exercise sequence

This exercise sequence introduces **while-loops** through a biological context around **gene expression, codons, and proteins**. The exercises show how a DN or RN sequence can be processed step by step using a counter.

The goal of the sequence is for students to learn how to use a **while** loop, update variables inside a loop, and stop a loop early with **break** when a certain condition is reached, such as finding a stop codon.

Assumed prior knowledge

Students are assumed to already be familiar with:

- variables;
- conditional statements such as **if** and **elif**;
- simple comparison operators such as **==**, **>**, **<**.

Topics within scope

- counter patterns;
- variables in loops;
- **break**.

The sequence builds up from processing a sequence base by base to reading codons in steps of three. By the end, students combine a counter with **break** to process a sequence until a stop codon is reached.

Topics outside the scope

- **continue**;
 - nested loops;
 - complex combinations of multiple loop structures.
-

Exercise 1: Reading DN bases one by one with a counter

Concepts

- while-loops
- indexing
- counter variables
- strings

Other Subject Concepts

- DN structure: bases
- Biology: genetics

Problem Description

In biology, DN consists of a sequence of bases: A (Adenine), T (Thymine), C (Cytosine), and G (Guanine). To understand genetic code, we need to be able to look at these bases one by one.

Write a program that goes through a string containing a DN sequence. Use a **while** loop and an index, or counter, to display each base separately on the screen.

Example input:

```
dna_sequence = " TGCT G"
```

Expected output:

```
T
G
C
T

G
```

Scaffolded Template

```
dna_sequence = " TGCT G"
```

```
index = 0
```

```
# The loop continues as long as the index is smaller than the length of the string
```

```
while index < len(dna_sequence):
```

```
    # TODO: Use the index to get the correct base from dna_sequence
```

```
    base =
```

```
    # TODO: Print the base to the screen
```

```
    # TODO: Increase the counter (index) by 1 so the loop moves to the next letter
```

Solution Code

```
dna_sequence = " TGCT G"
```

```
index = 0
```

```
while index < len(dna_sequence):
```

```
    # Get the current letter based on the position (index)
```

```
    base = dna_sequence[index]
```

```
    # Display the letter in the console
```

```
    print(base)
```

```
    # Increase the counter to reach the next position
```

```
    index = index + 1
```

Test Cases

```
## Test case
```

```
### Goal of the test case
```

This test case checks whether the student uses a ``while`` loop to print each base from the DN sequence ``TGCT G`` separately.

The expected output consists of seven lines: `` ``, ``T``, ``G``, ``C``, ``T``, `` ``, ``G``.

In addition, the test case checks whether the student uses a counter ``index`` and whether each base is temporarily stored in the variable ``base``.

```
### What is tested?
```

169

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

- Whether the student prints something with `print()` inside the while-loop.
- Whether the student prints exactly seven lines for the DN sequence `TGCT G`.
- Whether the student prints the correct base on each line.
- Whether the student uses the variable `index` as a counter in the while-loop.
- Whether the student correctly increases the counter `index` in each step.
- Whether the student assigns each base from the DN sequence to the variable `base`.
- Whether the variable `base` contains the last processed base after the loop.

Feedback the student may receive

- *Your code did not print anything. Use the print() function inside the while-loop.*
- *You printed {len(output)} lines, but we expected exactly {len(expected)} for the sequence TGCT G.*
- *Error on line {i+1}: You printed "{g}", but this should have been "{v}". Check whether you are using the correct index.*
- *You need to use the variable "index" as a counter in the while-loop.*
- *The counter "index" should be {len(dna_sequence)} at the end of the loop, but it is now {index}. Did you increase the counter in every step?*
- *You need to assign each base from the DN sequence to the variable "base".*
- *The last processed base should be "{dna_sequence[-1]}", but it is "{base}".*

Exercise 2: Counting specific DN bases with a limit

Concepts

- counter variable
- accumulator variable
- if-statement
- string indexing
- limit variable
- while-loop

Other Subject Concepts

- DN bases: Cytosine
- Sequence analysis

Problem Description

In the previous exercise, you learned how to use a counter to move step by step through a DN string. In biology, it is often important to analyze not the entire strand, but only a specific part, or fragment. In this exercise, we will count how often the base Cytosine (C) occurs at the beginning of a DN sequence.

Write a program that goes through the first 10 bases of a DN string. Use a limit variable to indicate that the loop must stop after 10 characters. Inside the loop, use an if-statement to check whether the current base is a C. If this is true, increase the accumulator variable.

Example input:

```
dna_sequence = "CGCT GCGTCG T"
limit = 10
```

Expected output:

The number of times C occurs is: 4

Hint: Think back to the previous exercise, where you used `counter = counter + 1` to navigate through the string. Now you use similar logic to keep track of the accumulator (`number_of_c`).

Scaffolded Template

```

dna_sequence = "CGCT GCGTCG T"
counter = 0
limit = 10
number_of_c = 0 # This is the accumulator we will use

# The loop stops when the counter reaches the limit
while counter < limit:
    # Get the base at the current position
    base = dna_sequence[counter]

    # TODO: Use an if-statement to check whether the base is equal to "C"
    # TODO: Increase the accumulator 'number_of_c' by 1 if the condition is true

    # TODO: Increase the counter by 1 to move to the next base

print("The number of times C occurs is:", number_of_c)

```

Solution Code

```

dna_sequence = "CGCT GCGTCG T"
counter = 0
limit = 10
number_of_c = 0

while counter < limit:
    base = dna_sequence[counter]

    if base == "C":
        number_of_c = number_of_c + 1

    counter = counter + 1

print("The number of times C occurs is:", number_of_c)

```

Test Cases

Test case

Goal of the test case

This test case checks whether the student counts the number of ``C`` bases within the first 10 characters of the DN sequence ``CGCT GCGTCG T``.

In those first 10 characters, exactly 4 ``C`` bases occur.

In addition, the test case checks whether the student uses the correct variables, whether the loop correctly stops at the limit, and whether the expected sentence is printed.

What is tested?

- Whether the student uses the variable ``number_of_c`` to keep track of the number of C bases.
- Whether the student uses the variable ``counter`` to go through the string.
- Whether the student correctly counts how many uppercase ``C`` bases occur within the first 10 characters.
- Whether the student makes the while-loop stop at the given limit ``10``.

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

- Whether the student does not continue counting until the end of the full DN sequence.
- Whether the student prints exactly the correct text.
- Whether the student increases the counter inside the loop.

Feedback the student may receive

- *`The code could not be executed because of a system error: {e}`*
- *`You did not use the variable `number_of_c` to keep track of the count.`*
- *`You did not use the variable `counter` to go through the string.`*
- *`The calculated number of C bases is {number\_of\_c}, but it should have been 4. Did you check for the uppercase letter C correctly?`*
- *`Your loop did not stop correctly at the limit. The counter is now {counter}, while the limit is {limit}. Use the condition `while counter < limit`.`*
- *`The printed text is not correct. Pay attention to spaces and capitalization. Expected: "{expected\_output}", but your code gave: "{output}"`*
- *`The counter was not increased inside the loop. Do not forget to add `counter = counter + 1`.`*

Note about the test case

The check for forgetting to increase the counter only works reliably if the automatic evaluation also uses a timeout. Without a timeout, an infinite loop may prevent the test case from continuing.

Exercise 3: DN codons: Jumping through the sequence

Concepts

- counter variable
- limit variable
- increment by 3
- string indexing
- while-loop with condition

Other Subject Concepts

- DN bases
- Codons: triplets

Problem Description

In the previous exercise, you learned how to use a counter to move step by step through a DN string by increasing the counter by 1 each time. In biology, however, DN is often read in groups of three, which we call codons.

In this exercise, we will look at the start position of each codon. To do this, we should not look at every base, but jump through the DN sequence in steps of three (index 0, 3, 6, and so on).

Write a program that goes through a DN string of 12 bases and prints the base that appears at the start of each codon. Use a limit of 12 to stop the loop.

Example input:

```
dna_sequence = " TGCGT CGTG "
```

Expected output:

```
Base at start position:
Base at start position: C
Base at start position:
Base at start position: T
```

Hint: In the previous exercise, you used `counter = counter + 1`. How can you adjust the counter to move 3 steps at a time?

Scaffolded Template

```

dna_sequence = " TGCCT CGTG "
counter = 0
limit = 12

while counter < limit:
    # TODO: Get the base at the current index (counter) from dna_sequence
    base =

    # TODO: Print the exact text "Base at start position: " followed by the found base
    print("Base at start position:", ...)

    # TODO: Increase the counter by 3 to jump to the next codon
    counter =

```

Solution Code

```

dna_sequence = " TGCCT CGTG "
counter = 0
limit = 12

while counter < limit:
    base = dna_sequence[counter]
    print("Base at start position:", base)
    counter = counter + 3

```

Test Cases

Test case

Goal of the test case

This test case checks whether the student uses a `while` loop to go through the DN sequence `TGCCT CGTG` in steps of 3.

The student must print the base at the start position of each codon.

The expected output consists of four lines:

```

`Base at start position: `, `Base at start position: C`, `Base at start position: `, `Base
at start position: T`.

```

What is tested?

- Whether the student does not use an index outside the DN sequence.
- Whether the student prints something inside the while-loop.
- Whether the student increases the counter in steps of 3 instead of steps of 1.
- Whether the student prints exactly four lines for a sequence of 12 bases.
- Whether the student prints the correct base at each start position of a codon.
- Whether the student prints the text exactly as expected, including spaces and wording.

Feedback the student may receive

- `IndexError: You are trying to get a base at an index outside the string. Did you set the limit of 12 correctly?`
- `n error occurred while executing your code: {e}`
- `Your code did not print anything. Do not forget to use the print() function inside the while-loop.`

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

- *`You printed 12 lines. This means you are looking at every base (steps of 1). The task is to jump by codon (steps of 3).`*
- *`You printed {len(output)} lines, but we expected exactly 4 (one for each codon in a sequence of 12 bases).`*
- *`Error on line {i+1}: we expected "{expected\_line}", but your code printed "{printed}". Pay attention to the spaces and the text!`*

Exercise 4: DNA stop codons: Ending the loop early

Concepts

- break-statement
- while-loop
- if-statement
- string slicing: 3 characters
- counter with step size 3

Other Subject Concepts

- Biology: DNA translation, codons, stop codons

Problem Description

In the previous exercise, you learned how to use a counter to jump through a DNA sequence in steps of 3. In biology, these groups of three bases are called codons. Some codons, such as T G, are stop codons. They indicate that the process of making a protein must stop immediately.

In this exercise, we will look at a DNA string. You use a while-loop to take 3 letters, one codon, at a time using string slicing. If the codon is T G, you must print a message and immediately end the loop with a **break** statement. If it is not a stop codon, print the codon and continue to the next one.

Example input:

```
dna_sequence = "GCT TCT GGGG"
```

Expected output:

```
Codon: GCT
```

```
Codon: TC
```

```
Stop codon T G found! We stop reading.
```

Hint: Use the logic from the previous exercise (`counter = counter + 3`) and combine it with an if-statement and the break command.

Scaffolded Template

```
dna_sequence = "GCT TCT GGGG"
```

```
counter = 0
```

```
limit = 12
```

```
while counter < limit:
```

```
    # TODO: Use string slicing to get 3 bases starting from the current counter
```

```
    codon = dna_sequence[counter:counter+3]
```

```
    # TODO: Use an if-statement to check whether the codon is equal to "T G"
```

```
    if codon == "T G":
```

```
        print("Stop codon T G found! We stop reading.")
```

```
        # TODO: add the break statement here to stop the loop
```

```
    # If it is not T G, we print the codon
```

```
    print("Codon:", codon)
```

TODO: Increase the counter by 3 to move to the next codon

Solution Code

```
dna_sequence = "GCT TCT GGGG"
counter = 0
limit = 12

while counter < limit:
    # Get the codon of 3 letters
    codon = dna_sequence[counter:counter+3]

    # Check for the stop codon
    if codon == "T G":
        print("Stop codon T G found! We stop reading.")
        break

    # Print the current codon
    print("Codon:", codon)

    # Increase the counter by 3 for the next codon
    counter = counter + 3
```

Test Cases

Test case

Goal of the test case

This test case checks whether the student processes a DN sequence per codon, meaning in steps of 3 bases at a time.

The student must print regular codons until the stop codon ``T G`` is found.

At the stop codon, the student must print a stop message and then stop reading.

What is tested?

- Whether the student produces output with ``print()``.
- Whether the student increases the counter by 3 each time.
- Whether the student does not print overlapping codons, such as ``CT ``, ``T T``, or ``TCT``.
- Whether the student prints the exact stop message when the stop codon ``T G`` is found.
- Whether the student stops after finding the stop codon.
- Whether the student does not print the stop codon ``T G`` as a regular codon.
- Whether the student prints the correct codons in the correct order.
- Whether the student prints exactly the expected number of lines.

Feedback the student may receive

- ``n error occurred while executing your code: {e}``
- ``Your code did not produce any output. Make sure you use print().``
- ``It looks like your counter is not increasing by 3 each time. You printed {line}.``
- ``The exact message "{stop_message}" was not found.``
- ``Your program did not stop after finding T G. Did you use the break statement?``
- ``You may not print the stop codon "T G" as a regular codon. Use an if-statement to prevent this.``
- ``You printed {len(lines)} lines, but we expected exactly {len(expected_output)}.`` 175

```
- `Error on line {i+1}: Expected "{expected}", but got "{actual}"/>.
```

```
### Note about the test case
```

The student's code is executed twice in this test case.

Exercise 5: Translation: Translating a DN sequence

Concepts

- counter variable
- limit variable
- increment: step size 3
- break
- if-statement
- string slicing: 3 characters

Other Subject Concepts

- Biology: translation
- Codons and stop codons

Problem Description

In the previous exercises, you learned how to move through a DN string in steps of 3 and how to use `break` to stop a loop. In this exercise, we combine these techniques for a simulation of translation.

Although translation in biology normally takes place on mRNA, in this model we use the DN coding strand, which has almost the same sequence as the mRNA. You are given a DN sequence of 12 letters. You must write a program that looks at this sequence three letters at a time, or one codon at a time. The loop must stop in two cases: 1. The counter reaches the limit of 12, which is the end of the sequence. 2. The codon T G is found. If this happens, use a `break` to end the loop immediately.

Example input:

```
dna_sequence = "GCT TCT GGGG"
limit = 12
```

Expected output:

```
Codon: GCT
Codon: TC
Stop codon T G found! The simulation stops.
```

Hint: Use what you learned in the previous exercise. Take 3 characters each time with `dna_sequence[counter:counter+3]` and do not forget to increase the counter by 3 at the end of the loop.

Solution Code

```
dna_sequence = "GCT TCT GGGG"
counter = 0
limit = 12

while counter < limit:
    # Use string slicing to get the current codon (3 letters)
    codon = dna_sequence[counter:counter+3]

    # Use an if-statement to check whether the codon is equal to 'T G'
    if codon == "T G":
        print("Stop codon T G found! The simulation stops.")
        # Stop the loop immediately
        break

    counter += 3
```

```

    bre k

    # Print the current codon
    print("Codon:", codon)

    # Increase the counter by 3 to move to the next codon
    counter = counter + 3

```

Test Cases

Test case

Goal of the test case

This test case checks whether the student correctly processes a DN sequence per codon, meaning in steps of 3 bases at a time.

The test case uses multiple DN sequences to check whether the solution works not only for the example in the task.

It tests whether the student prints regular codons, correctly stops at the stop codon ``T G``, and gives the correct output in different situations.

What is tested?

- Whether the student does not cause an error while the code is being executed.
- Whether the student processes the DN sequence in groups of 3 bases.
- Whether the student prints regular codons in the form ``Codon: ...``.
- Whether the student stops when the stop codon ``T G`` is found.
- Whether the student prints the correct stop message at the stop codon.
- Whether the student correctly handles a DN sequence without a stop codon.
- Whether the student correctly stops when the stop codon appears directly at the beginning.
- Whether the student correctly stops when the stop codon appears at the last position.
- Whether the student prints the correct number of lines.
- Whether the student prints the expected lines in the correct order.

Feedback the student may receive

```

- `Error in test '{test_name}': Your code caused an error: {e}`
- `Error in test '{test_name}': The number of printed lines is incorrect.
Expected: {len(expected_output)} lines, Got: {len(actual_lines)} lines.
Your output was:
{actual_lines}`
- `Error in test '{test_name}' on line {i+1}:
Expected: '{expected}'
Got: '{actual}'`

```

C.5.1 Dutch

Generated Exercises

Introductie van de oefeningenreeks

Deze oefeningenreeks introduceert **while-lussen** aan de hand van een biologische context rond **genexpressie, codons en eiwitten**. De oefeningen tonen hoe een DN - of RN -sequentie stap voor stap kan worden doorlopen met een teller.

Het doel van de reeks is dat leerlingen leren om een **while**-lus te gebruiken, variabelen binnen een lus bij te werken en een lus vroegtijdig te stoppen met **break** wanneer een bepaalde voorwaarde bereikt is, zoals het vinden van een stopcodon.

Veronderstelde voorkennis

Leerlingen worden verondersteld al vertrouwd te zijn met:

- variabelen;
- conditionele instructies zoals **if** en **elif**;
- eenvoudige vergelijkingsoperatoren zoals **==**, **>**, **<**.

Onderwerpen binnen de scope

- tellerpatronen;
- variabelen in loops;
- **break**.

De reeks bouwt op van het doorlopen van een sequentie per base naar het lezen van codons in stappen van drie. Op het einde combineren leerlingen een teller met **break** om een sequentie te verwerken tot aan een stopcodon.

Onderwerpen buiten de scope

- **continue**;
- geneste lussen;
- complexe combinaties van meerdere lusstructuren.

Exercise 1: DN -basen één voor één lezen met een teller

Concepts

- while-loops
- indexing
- teller-variabelen
- strings

Other Subject Concepts

- DN -structuur (basen)
- Biologie (genetica)

Problem Description

In de biologie bestaat DN uit een reeks basen: (denine), T (Thymine), C (Cytosine) en G (Guanine). Om de genetische code te begrijpen, moeten we deze basen één voor één kunnen bekijken.

179

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

Schrijf een programma dat een string met een DN -sequentie doorloopt. Gebruik een 'while-loop' en een index (een teller) om elke base afzonderlijk op het scherm te tonen.

Voorbeeld van invoer: dna_sequentie = ' TGCT G'

Verwachte uitvoer: T G C T G

Scaffolded Template

```
dna_sequentie = " TGCT G"
index = 0

# De loop gaat door zolang de index kleiner is dan de lengte van de string
while index < len(dna_sequentie):
    # TODO: Gebruik de index om de juiste base uit de dna_sequentie te halen
    base =

    # TODO: Print de base naar het scherm

    # TODO: Verhoog de teller (index) met 1 zodat de loop naar de volgende letter gaat
```

Solution Code

```
dna_sequentie = " TGCT G"
index = 0

while index < len(dna_sequentie):
    # Haal de huidige letter op basis van de positie (index)
    base = dna_sequentie[index]

    # Toon de letter in de console
    print(base)

    # Verhoog de teller om de volgende positie te bereiken
    index = index + 1
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling met een `while`-loop elke base uit de DN -sequentie ` TGCT G` afzonderlijk print.

De verwachte uitvoer bestaat uit zeven regels: ` `, `T`, `G`, `C`, `T`, ` `, `G`.

Daarnaast controleert de testcase of de leerling een teller `index` gebruikt en of elke base tijdelijk wordt opgeslagen in de variabele `base`.

Wat wordt getest?

- Of de leerling iets print met `print()` binnen de while-loop.
- Of de leerling exact zeven regels print voor de DN -sequentie ` TGCT G`.
- Of de leerling de juiste base print op elke regel.
- Of de leerling de variabele `index` gebruikt als teller in de while-loop.
- Of de leerling de teller `index` in elke stap correct verhoogt.

- Of de leerling elke base uit de DN -sequentie toekent aan de variabele ``base``.
- Of de variabele ``base`` na de loop de laatst verwerkte base bevat.

Feedback die de leerling kan krijgen

- *`Je code heeft niets geprint. Gebruik de print() functie binnen de while-loop.`*
- *`Je hebt {len(output)} regels geprint, maar we verwachtten er exact {len(verwacht)} voor de sequentie TGCT G.`*
- *`Fout op regel {i+1}: Je printte "{g}", maar dit moest "{v}" zijn. Controleer of je de juiste index gebruikt.`*
- *`Je moet de variabele "index" gebruiken als teller in de while-loop.`*
- *`De teller "index" zou aan het einde van de loop {len(dna\_sequentie)} moeten zijn, maar is nu {index}. Heb je de teller in elke stap verhoogd?`*
- *`Je moet elke base uit de DN -sequentie toekennen aan de variabele "base".`*
- *`De laatst verwerkte base zou "{dna\_sequentie[-1]}" moeten zijn, maar is "{base}".`*

Exercise 2: Specifieke DN -basen tellen met een limiet

Concepts

- teller-variabele
- accumulator-variabele
- if-statement
- string-indexering
- limiet-variabele
- while-loop

Other Subject Concepts

- DN -basen (Cytosine)
- Sequentie-analyse

Problem Description

In de vorige oefening heb je geleerd hoe je met een teller stap voor stap door een DN -string kunt lopen. In de biologie is het vaak belangrijk om niet de hele streng, maar slechts een specifiek deel (een fragment) te analyseren. In deze oefening gaan we tellen hoe vaak de base Cytosine (C) voorkomt in het begin van een DN -sequentie.

Schrijf een programma dat de eerste 10 basen van een DN -string doorloopt. Gebruik een limiet-variabele om aan te geven dat de loop moet stoppen na 10 tekens. Gebruik binnen de loop een if-statement om te controleren of de huidige base een 'C' is. Is dit waar is, verhoog je de accumulator-variabele.

Voorbeeld van invoer: `dna_sequentie = 'CGCT GCGTCG T'` limiet = 10

Verwachte uitvoer: Het aantal keer dat C voorkomt is: 4

Hint: Denk aan de vorige oefening waarbij je `teller = teller + 1` gebruikte om door de string te navigeren. Nu gebruik je een soortgelijke logica om de accumulator (`aantal_c`) bij te houden.

Scaffolded Template

```
dna_sequentie = "CGCT GCGTCG T"
teller = 0
limiet = 10
aantal_c = 0 # Dit is de accumulator die we gaan gebruiken

# De loop stopt als de teller de limiet bereikt
while teller < limiet:
    # Haal de base op de huidige positie op
    base = dna_sequentie[teller]
```

181

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

```
# TODO: Gebruik een if-statement om te kijken of de base gelijk is aan "C"
# TODO: Verhoog de accumulator 'aantal_c' met 1 als de voorwaarde waar is

# TODO: Verhoog de teller met 1 om naar de volgende base te gaan
```

```
print("Het aantal keer dat C voorkomt is:", aantal_c)
```

Solution Code

```
dna_sequentie = "CGCT GCGTCG T"
teller = 0
limiet = 10
aantal_c = 0

while teller < limiet:
    base = dna_sequentie[teller]

    if base == "C":
        aantal_c = aantal_c + 1

    teller = teller + 1

print("Het aantal keer dat C voorkomt is:", aantal_c)
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling het aantal `C`-basen telt binnen de eerste 10 tekens van de DN -sequentie `CGCT GCGTCG T`.

In die eerste 10 tekens komen precies 4 `C`-basen voor.

Daarnaast controleert de testcase of de leerling de juiste variabelen gebruikt, de loop correct laat stoppen bij de limiet, en de verwachte zin print.

Wat wordt getest?

- Of de leerling de variabele `aantal\_c` gebruikt om het aantal C-basen bij te houden.
- Of de leerling de variabele `teller` gebruikt om door de string te lopen.
- Of de leerling correct telt hoeveel hoofdletters `C` voorkomen binnen de eerste 10 tekens.
- Of de leerling de while-loop laat stoppen bij de opgegeven limiet `10`.
- Of de leerling niet verder telt tot het einde van de volledige DN -sequentie.
- Of de leerling exact de juiste tekst print.
- Of de leerling de teller binnen de loop verhoogt.

Feedback die de leerling kan krijgen

- `De code kon niet worden uitgevoerd door een systeemfout: {e}`
- `Je hebt de variabele aantal\_c niet gebruikt om de teller bij te houden.`
- `Je hebt de variabele teller niet gebruikt om door de string te lopen.`
- `Het berekende aantal C-basen is {aantal\_c}, maar dit had 4 moeten zijn. Heb je goed gecontroleerd op de hoofdletter C?`

C.5. While loop exercise sequence

- ``Je loop is niet correct gestopt bij de limiet. De teller staat nu op {teller}, terwijl de limiet {limiet} is. Gebruik de voorwaarde while teller < limiet.``
- ``De geprinte tekst is niet correct. Let goed op spaties en hoofdletters. Verwacht: "{verwachte_output}", maar jouw code gaf: "{output}"``
- ``De teller is niet verhoogd binnen de loop. Vergeet niet teller = teller + 1 toe te voegen.``

Opmerking bij de testcase

De controle op het vergeten verhogen van de teller werkt alleen betrouwbaar als de automatische evaluatie ook een timeout gebruikt. Zonder timeout kan een oneindige loop ervoor zorgen dat de testcase niet verder uitgevoerd wordt.

Exercise 3: DN -codons: Springen door de sequentie

Concepts

- teller-variabele
- limiet-variabele
- increment van 3
- string-indexering
- while-loop met conditie

Other Subject Concepts

- DN -basen
- Codons (triplet)

Problem Description

In de vorige oefening heb je geleerd hoe je met een teller stap voor stap door een DN -string kunt lopen door de teller telkens met 1 te verhogen. In de biologie wordt DN echter vaak gelezen in groepjes van drie, die we codons noemen.

In deze oefening gaan we de startpositie van elk codon bekijken. Om dit te doen, moeten we niet elke base bekijken, maar met sprongen van drie door de DN -sequentie lopen (index 0, 3, 6, enzovoort).

Schrijf een programma dat een DN -string van 12 basen doorloopt en telkens de base print die aan het begin van een codon staat. Gebruik een limiet van 12 om de loop te stoppen.

Voorbeeld van invoer: `dna_sequentie = ' TGC GT CGTG '`

Verwachte uitvoer: Base op startpositie: Base op startpositie: C Base op startpositie: Base op startpositie: T

Hint: In de vorige oefening gebruikte je `teller = teller + 1`. Hoe kun je de teller aanpassen om steeds 3 stappen tegelijk te zetten?

Scaffolded Template

```
dna_sequentie = " TGC GT CGTG "
teller = 0
limiet = 12

while teller < limiet:
    # TODO: Haal de base op de huidige index (teller) uit de dna_sequentie
    base =

    # TODO: Print de exacte tekst "Base op startpositie: " gevolgd door de gevonden base
    print("Base op startpositie:", ...)

    # TODO: Verhoog de teller met 3 om naar het volgende codon te springen
    teller =
```

183

Solution Code

```
dna_sequentie = " TGCCT CGTG "  
teller = 0  
limiet = 12  
  
while teller < limiet:  
    base = dna_sequentie[teller]  
    print("Base op startpositie:", base)  
    teller = teller + 3
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling met een `while`-loop door de DN -sequentie `TGCCT CGTG` loopt in stappen van 3.

De leerling moet telkens de base op de startpositie van elk codon printen.

De verwachte uitvoer bestaat uit vier regels:

```
`Base op startpositie: `, `Base op startpositie: C`, `Base op startpositie: `, `Base op  
startpositie: T`.
```

Wat wordt getest?

- Of de leerling geen index gebruikt die buiten de DN -sequentie valt.
- Of de leerling iets print binnen de while-loop.
- Of de leerling de teller verhoogt met stappen van 3 in plaats van stappen van 1.
- Of de leerling precies vier regels print voor een sequentie van 12 basen.
- Of de leerling de juiste base op elke startpositie van een codon print.
- Of de leerling de tekst exact print zoals verwacht, inclusief spaties en formulering.

Feedback die de leerling kan krijgen

- ``IndexError: Je probeert een base op te halen op een index die buiten de string valt. Heb je de limiet van 12 goed ingesteld?``
- ``Er is een fout opgetreden tijdens het uitvoeren van je code: {e}``
- ``Je code heeft niets geprint. Vergeet niet de print() functie te gebruiken binnen de while-loop.``
- ``Je hebt 12 regels geprint. Dit betekent dat je elke base bekijkt (stapjes van 1). De opdracht is om per codon te springen (stapjes van 3).``
- ``Je hebt {len(output)} regels geprint, maar we verwachtten er precies 4 (één voor elk codon in een sequentie van 12 basen).``
- ``Fout op regel {i+1}: we verwachtten "{verwachte_regel}", maar jouw code printte "{geprint}"`.
Let goed op de spaties en de tekst!``

Exercise 4: DN Stopcodons: De loop vroegtijdig beëindigen

Concepts

- break-statement
- while-loop
- if-statement
- string-slicing (3 karakters)
- teller met stapgrootte 3

Other Subject Concepts

- Biologie (DN -translatie, codons, stopcodons)

Problem Description

In de vorige oefening heb je geleerd hoe je met een teller in stappen van 3 door een DN -sequentie kunt ‘springen’. In de biologie worden deze groepjes van drie basen ‘codons’ genoemd. Sommige codons, zoals ‘T G’, zijn stopcodons. Ze geven aan dat het proces van eiwit maken direct moet stoppen.

In deze oefening gaan we een DN -string bekijken. Je gebruikt een while-loop om telkens 3 letters (een codon) tegelijk te pakken met string-slicing. Is het codon ‘T G’ is, moet je een melding printen en de loop direct beëindigen met een ‘break’-statement. Is het geen stopcodon is, print je het codon en ga je door naar de volgende.

Voorbeeld van invoer: dna_sequentie = ‘GCT TCT GGGG’

Verwachte uitvoer: Codon: GCT Codon: TC Stopcodon T G gevonden! We stoppen met lezen.

Hint: Gebruik de logica uit de vorige oefening (teller = teller + 3) en combineer dit met een if-statement en het break-commando.

Scaffolded Template

```
dna_sequentie = "GCT TCT GGGG"
teller = 0
limiet = 12

while teller < limiet:
    # TODO: Gebruik string-slicing om 3 basen te pakken vanaf de huidige teller
    codon = dna_sequentie[teller:teller+3]

    # TODO: Controleer met een if-statement of het codon gelijk is aan "T G"
    if codon == "T G":
        print("Stopcodon T G gevonden! We stoppen met lezen.")
        # TODO: Voeg hier het break-statement toe om de loop te stoppen

    # Is het geen T G is, printen we het codon
    print("Codon:", codon)

    # TODO: Verhoog de teller met 3 om naar het volgende codon te gaan
```

Solution Code

```
dna_sequentie = "GCT TCT GGGG"
teller = 0
limiet = 12

while teller < limiet:
    # Haal het codon van 3 letters op
    codon = dna_sequentie[teller:teller+3]

    # Controleer op het stopcodon
    if codon == "T G":
        print("Stopcodon T G gevonden! We stoppen met lezen.")
        break

    # Print het huidige codon
    print("Codon:", codon)

    # Verhoog de teller met 3 voor het volgende codon
```

```
teller = teller + 3
```

Test Cases

Testcase

Doel van de testcase

Deze testcase controleert of de leerling een DN -sequentie per codon verwerkt, dus telkens in stappen van 3 basen.

De leerling moet gewone codons printen tot het stopcodon ``T G`` wordt gevonden.

Bij het stopcodon moet de leerling een stopmelding printen en daarna stoppen met lezen.

Wat wordt getest?

- Of de leerling uitvoer produceert met ``print()``.
- Of de leerling de teller telkens met 3 verhoogt.
- Of de leerling geen overlappende codons print, zoals ``CT ``, ``T T`` of ``TCT``.
- Of de leerling de exacte stopmelding print wanneer het stopcodon ``T G`` wordt gevonden.
- Of de leerling stopt na het vinden van het stopcodon.
- Of de leerling het stopcodon ``T G`` niet print als een gewoon codon.
- Of de leerling de juiste codons in de juiste volgorde print.
- Of de leerling precies het verwachte aantal regels print.

Feedback die de leerling kan krijgen

- *`Er trad een fout op tijdens het uitvoeren van je code: {e}`*
- *`Je code heeft geen uitvoer geproduceerd. Zorg dat je print() gebruikt.`*
- *`Het lijkt erop dat je teller niet met 3 tegelijk ophoogt. Je hebt {line} geprint.`*
- *`De exacte melding "{stop\_melding}" werd niet gevonden.`*
- *`Je programma is niet gestopt na het vinden van T G. Heb je het break-statement gebruikt?`*
- *`Je mag het stopcodon "T G" niet printen als een gewoon codon. Gebruik een if-statement om dit te voorkomen.`*
- *`Je hebt {len(lines)} regels geprint, maar we verwachtten er precies {len(verwachte uitvoer)}.`*
- *`Fout op regel {i+1}: Verwachtte "{verwacht}", maar kreeg "{werkelijk}"`.*

Opmerking bij de testcase

De code van de leerling wordt in deze testcase twee keer uitgevoerd.

Exercise 5: Translatie: Een DN -sequentie vertalen

Concepts

- teller-variabele
- limiet-variabele
- increment (stapgrootte 3)
- break
- if-statement
- string-slicing (3 karakters)

Other Subject Concepts

- Biologie (Translatie)
- Codons en stopcodons

Problem Description

In de vorige oefeningen heb je geleerd hoe je met sprongen van 3 door een DN -string kunt lopen en hoe je een 'break' gebruikt om een loop te stoppen. In deze oefening combineren we deze technieken voor een simulatie van translatie.

Hoewel translatie in de biologie normaal gesproken op mRNA plaatsvindt, gebruiken we in dit model de DN -coderende streng (die nagenoeg dezelfde volgorde heeft als het mRNA). Je krijgt een DN -sequentie van 12 letters. Je moet een programma schrijven dat deze sequentie per drie letters (een codon) bekijkt. De loop moet stoppen in twee gevallen: 1. De teller bereikt de limiet van 12 (het einde van de sequentie). 2. Het codon 'T G' wordt gevonden. Als dit gebeurt, gebruik je een 'break' om de loop onmiddellijk te beëindigen.

Voorbeeld van invoer: dna_sequentie = 'GCT TCT GGGG' limiet = 12

Verwachte uitvoer: Codon: GCT Codon: TC Stopcodon T G gevonden! De simulatie stopt.

Hint: Gebruik de kennis uit de vorige oefening. Pak telkens 3 karakters met dna_sequentie[teller:teller+3] en vergeet niet de teller aan het einde van de loop met 3 te verhogen.

Solution Code

```
dna_sequentie = 'GCT TCT GGGG'
teller = 0
limiet = 12

while teller < limiet:
    # Gebruik string-slicing om het huidige codon (3 letters) te pakken
    codon = dna_sequentie[teller:teller+3]

    # Gebruik een if-statement om te controleren of het codon gelijk is aan 'T G'
    if codon == 'T G':
        print('Stopcodon T G gevonden! De simulatie stopt.')
        # Stop de loop onmiddellijk
        break

    # Print het huidige codon
    print('Codon:', codon)

    # Verhoog de teller met 3 om naar het volgende codon te gaan
    teller = teller + 3
```

Test Cases

```
## Testcase
```

```
### Doel van de testcase
```

Deze testcase controleert of de leerling een DN -sequentie correct per codon verwerkt, dus telkens in stappen van 3 basen.

De testcase gebruikt meerdere DN -sequenties om te controleren of de oplossing niet alleen werkt voor het voorbeeld uit de opdracht.

Er wordt getest of de leerling gewone codons print, correct stopt bij het stopcodon 'T G', en de juiste uitvoer geeft in verschillende situaties.

```
### Wat wordt getest?
```

- Of de leerling geen fout veroorzaakt tijdens het uitvoeren van de code.
- Of de leerling de DN -sequentie verwerkt in groepen van 3 basen.

187

C. TEACHER AND DOMAIN-EXPERT EVALUATION INSTRUMENTS

- Of de leerling gewone codons print in de vorm ``Codon: ...``.
- Of de leerling stopt wanneer het stopcodon ``T G`` wordt gevonden.
- Of de leerling de juiste stopmelding print bij het stopcodon.
- Of de leerling correct omgaat met een DN -sequentie zonder stopcodon.
- Of de leerling correct stopt wanneer het stopcodon direct aan het begin staat.
- Of de leerling correct stopt wanneer het stopcodon op de laatste positie staat.
- Of de leerling het juiste aantal regels print.
- Of de leerling de verwachte regels in de juiste volgorde print.

Feedback die de leerling kan krijgen

```
- `Fout bij test '{test_name}': Je code veroorzaakte een error: {e}`
- `Fout in test '{test_name}': Het aantal geprinte regels is onjuist.
Verwacht: {len(expected_output)} regels, Gekregen: {len(actual_lines)} regels.
Jouw uitvoer was:
{actual_lines}`
- `Fout in test '{test_name}' op regel {i+1}:
Verwacht: '{expected}'
Gekregen: '{actual}'`
```

C.6 Teacher Follow-Up Interview Guide

Background and Current Practice

- How much freedom do you have in selecting and ordering programming exercises?
- How do you currently prepare programming lessons?
 - Do you use existing exercises from the web or other sources?
 - Do you create your own exercises?
 - How do you combine exercises into a coherent lesson or exercise sequence?
- How could a system that generates programming exercises fit into your current lesson-preparation workflow?
- Would generated exercises help you when preparing programming lessons? Why or why not?

Evaluation of the Generated Exercises

- How would you evaluate the generated exercises overall?
- What are the main strengths of the exercises?
- What are the main issues or limitations of the exercises?
- Are there any aspects of the exercises that are incorrect or inappropriate?
- Would you like to see more variation in the types of exercises that are generated?
- How should different exercise types be combined within an exercise sequence?

Exercise Components

If useful, the discussion can consider the different components of an exercise separately:

- Metadata
- Problem statement and template code
- Model solution
- Test cases and feedback

Possible follow-up questions include:

- Which components of the exercises work well?

- Which components would you change?
- Is any information currently missing from the exercises?
- Would references to related or previous exercises be useful in the metadata?
- Would additional information for teachers, such as integration hints, be useful?
- Once a complete exercise has been generated, which parts should the teacher be able to edit or regenerate separately?
- How much editing would be acceptable before the generated material stops saving you time?

Contextualization and Student Use

- Which aspects of the generated exercises do you think would work well for students?
- Are the problem descriptions sufficiently clear?
- Is the amount of information provided appropriate?
- Are there parts of the problem descriptions that could be reduced, removed, rewritten, or represented differently?
- Could visual material, such as figures or images, be useful in some exercises?
- How well is the biological context integrated into the programming exercises?
- What works well about the contextualization?
- What could be improved about the contextualization?
- Should teachers be able to choose how strongly the context is integrated into an exercise?

Usability and Curriculum Integration

- How usable is the generated material in its current form?
- Could using generated exercises save time compared with how you currently source or create exercises?
- How difficult would it be to integrate these exercises into the existing curriculum?
- How could an exercise-sequence generation system best be integrated into your workflow?
- What functionality would make such a system easier or more useful to integrate into your workflow?

Teacher Control and Sequence Generation

- What kind of control should teachers have over the generation process?
- Should teachers be able to specify which types of exercises should appear in a sequence?
- Would you prefer the system to generate a complete exercise sequence, or provide a larger set of exercises from which the teacher can select?
- Would you prefer to receive only the final generated sequence, or would it be useful to be involved during the generation process?
- If teachers could steer the system during generation, how should this work?
- Would you prefer to approve exercises individually or review the complete sequence at the end?
- If one exercise is changed, should the system automatically reconsider later exercises in the sequence?

Test Cases

- What did you think of the generated test cases?
- Do the test cases adequately assess the intended programming concepts?
- Is the feedback produced by the tests useful for students?
- What should teachers be able to control or modify about the generated tests?

Future Development

- What changes would make the generator more useful for you?
- What additional forms of teacher control would you like to have?
- Are there any important features or types of generated material that are currently missing?

ppendix D

Supplementary Results of the exploratory interviews and teachers survey

This appendix provides supplementary results from the exploratory teacher survey and interviews that support the findings discussed in Section ??.

D.1 Supplementary Teacher Survey Results

Figure ?? shows the ratings for progressively structured exercises across four broad programming categories. Table D.1 summarizes the conditions identified in the open survey responses.

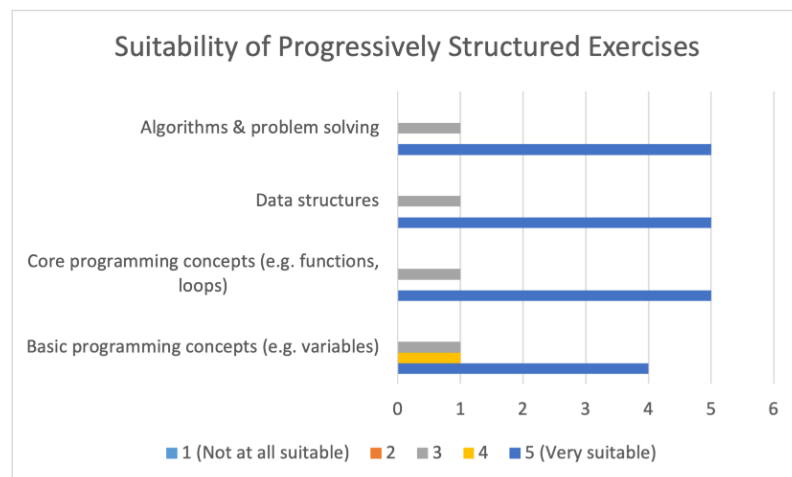


FIGURE D.1: Results of the teacher survey on the perceived suitability of progressively structured exercise sequences across broad programming categories.

D. SUPPLEMENTARY RESULTS OF THE EXPLORATORY INTERVIEWS AND TEACHERS SURVEY

Identified condition	Participants
Meaningful and recognizable interdisciplinary connection	P2, P3, P4
Alignment between biology and informatics learning goals/-content	P3, P6
Biological content should be familiar to students	P3
Feasibility within the available lesson time	P3
Teacher collaboration and support	P2, P5, P6
Availability of teaching materials and concrete examples	P6

TABLE D.1: Conditions for successful implementation of interdisciplinary programming exercises identified in the open survey responses. P1 did not provide a response to this question.

D.2 Exploratory Interview Findings

Table D.2 summarizes the main recurring themes identified across the three exploratory interviews.

Theme	Summary from the interviews
Organization of informatics education	The implementation of informatics differs between schools and depends on the educational setting, available teaching hours, teacher expertise, infrastructure, and whether informatics is taught separately or integrated into STEM/science.
Need for usable teaching material	Developing suitable material from scratch requires considerable teacher effort. Participants indicated a need for ready-to-use material that can still be adapted to the classroom, student level, and teacher approach.
Gradual progression	Introductory programming should be built up gradually, starting with simple or guided activities before moving towards more independent programming and larger synthesis tasks.
Support for beginning programmers	Students can struggle with basic concepts, combining several concepts, translating a problem into programming steps, debugging, and differences in prior knowledge and pace.
Contextualization	Scientific or biological contexts can make the purpose and application of programming more visible, particularly when they are recognizable and connected to students' existing knowledge.
Conditions for contextualization	The context should remain understandable, scientifically correct, and clearly connected to the programming goal. It should not introduce unnecessary complexity or require excessive additional domain knowledge.
Teacher control and support	Teachers need control over the level, content, progression, and material they use. Model solutions, testing possibilities, differentiation, examples, and support for less experienced teachers were identified as useful.

TABLE D.2: Summary of the main themes identified across the three exploratory interviews. Interview findings are based on written notes and are therefore presented as paraphrased themes rather than direct quotations.

ppendix E

eTeacher representation

The eTeacher platform represents course material in YAML files. The platform supports multiple activity types, but the generation system developed in this thesis targets programming exercises. This appendix therefore only describes the fields and test structure that are relevant to the generated programming exercises.

E.1 Relevant programming-exercise fields

Table E.1 summarizes the main fields used to represent the generated programming exercises in eTeacher. The platform supports additional optional fields, but these are omitted here because they were not central to the generated material used in this study.

Field	Purpose
<code>type</code>	Identifies the activity as a programming exercise.
<code>name</code>	Defines the title of the programming exercise.
<code>content_items</code>	Contains the student-facing problem description and instructions.
<code>template</code>	Contains the starter code provided to the student.
<code>solution</code>	Contains the reference solution used for validation and teacher review.
<code>script</code>	Determines whether the exercise is executed as script-level code or as a function-based exercise.
<code>test</code>	Contains the automated tests used to evaluate the student solution.

TABLE E.1: Main eTeacher fields relevant to the generated programming exercises.

E.2 Test representation used in the generated exercises

The generated exercises used in this study were primarily represented as script-based exercises using `script: true`. Their automated evaluation therefore relied mainly on output tests. These tests execute the student’s code and compare the resulting output with the expected output. Although eTeacher supports additional test types, these are not discussed here because they were not central to the generated exercise sequence used in this study. Table E.2 summarizes the fields used in this test representation.

Field	Purpose
<code>type</code>	Defines the type of test. For the generated exercises considered here, <code>output</code> tests were used.
<code>name</code>	Provides a descriptive name for the test.
<code>expected_output</code>	Specifies the output expected from a correct solution.
<code>alternative_outputs</code>	Optionally specifies alternative outputs that are also accepted.
<code>random_seed</code>	Optionally fixes the random seed when deterministic behaviour is required.
<code>output_transformations</code>	Optionally specifies transformations applied to the produced output before comparison.

TABLE E.2: Relevant fields of the eTeacher output-test representation used in the generated exercises.

ppendix F

Examples from the Design based System Development

This appendix provides selected examples that support the iterative development described in Section 4.2. The examples were selected to illustrate the main capabilities introduced during development and the limitations that motivated subsequent architectural changes. Only the components relevant to each observation are shown; the complete generation logs are available in the project repository.

F.1 Baseline and Modular Generation Examples

F.1.1 Baseline Generation

```

## Guess the Secret Number!

**Concepts Used:**
* **Programming:** 'while loops', 'variables', 'input()', 'print()', 'if/elif/else', 'integer conversion', 'random module', 'boolean logic'
* **Other Subjects:** arithmetic, 'logic', 'comparison'
Not need for this exercise

**Task Explanation:**
Your goal is to create a simple "Guess the Number" game. The computer will randomly select a whole number between 1 and 10. The player then tries to guess this number. After each guess, the computer will tell the player if their guess was too high, too low, or correct. The game continues until the player guesses the correct number.

**Python Code Solution:**

'''python
import random

def guess_the_number_game():
    """
    Plays a simple "Guess the Number" game.
    The computer picks a random number, and the user guesses until correct.
    """

    # 1. The computer picks a secret number between 1 and 10.
    secret_number = random.randint(1, 10)
    print("I'm thinking of a number between 1 and 10.")
    print("Can you guess what it is?")

    # 2. Initialize a flag to control the loop.
    guessed_correctly = False

    # 3. Start the game loop. It continues as long as the number hasn't been guessed.
    while not guessed_correctly:
        try:
            # 4. Get the player's guess as input.
            guess_str = input("Enter your guess: ")
            guess = int(guess_str) # Convert the input string to an integer

            # 5. Check if the guess is too low, too high, or correct.
            if guess < secret_number:
                print("Your guess is too low. Try again!")
            elif guess > secret_number:
                print("Your guess is too high. Try again!")
            else:
                # 6. If the guess is correct, congratulate the player and end the loop.
                print("Congratulations! You guessed the secret number, which was {secret_number}!")
                guessed_correctly = True # Set the flag to True to exit the while loop
        except ValueError:
            # Handle cases where the input is not a valid number.
            print("That's not a valid number. Please enter a whole number.")

    # To run the game:
    # guess_the_number_game()

```

FIGURE F.1: What a LLM can generate without much guidance

Test Cases:

Since the `secret_number` is random, we describe scenarios rather than exact input/output.

- Guessing Correctly on the First Try:**
 - Scenario: The computer randomly picks `5`.
 - User Input: `5`
 - Expected Output: `Congratulations! You guessed the secret number, which was 5! (The game ends.)`
- Guessing Too Low, Then Too High, Then Correct:**
 - Scenario: The computer randomly picks `7`.
 - User Input:
 - `3`
 - `9`
 - `7`
 - Expected Output:
 - `Your guess is too low. Try again!`
 - `Your guess is too high. Try again!`
 - `Congratulations! You guessed the secret number, which was 7! (The game ends.)`
- Guessing Too High, Then Too Low, Then Correct:**
 - Scenario: The computer randomly picks `4`.
 - User Input:
 - `8`
 - `2`
 - `4`
 - Expected Output:
 - `Your guess is too high. Try again!`
 - `Your guess is too low. Try again!`
 - `Congratulations! You guessed the secret number, which was 4! (The game ends.)`
- Entering Invalid Input:**
 - Scenario: The computer picks any number.
 - User Input:
 - `hello`
 - `6` (assuming `6` is the correct guess this time)
 - Expected Output:
 - `That's not a valid number. Please enter a whole number.`
 - `Congratulations! You guessed the secret number, which was 6! (The game continues after invalid input until a valid, correct guess is made.)`

FIGURE F.2: Limitation: Not python test cases

F.1.2 Static prompt

```
{
  "title": "Simple Counter with Loops",
  "concepts": [
    "for loop",
    "range function",
    "print function",
    "variables"
  ],
  "other_subject_concepts": [],
  "problem_description": "Write a Python program that uses a 'for' loop to count from 1 up to a given number (inclusive) and prints each number on a new line. The program should take an integer as input for the upper limit of the count.",
  "python_solution_code": "def simple_counter(limit):\n    \"\"\"Counts from 1 up to the given limit and prints each number.\"\"\"\n    Args:\n        limit (int): The upper bound for the counting.\n    if not isinstance(limit, int) or limit < 1:\n        print(\"Input must be a positive integer.\")\n    return\n    for number in range(1, limit + 1):\n        print(number)\n\n# Example usage:\nsimple_counter(5)",
  "python_test_cases": "import io\nimport sys\n\ndef test_simple_counter():\n    # Test case 1: Basic functionality\n    print(\"--- Test Case 1: Basic functionality ---\")\n    captured_output = io.StringIO()\n    sys.stdout = captured_output\n    simple_counter(5)\n    sys.stdout = sys.__stdout__\n    # Reset redirect\n    output = captured_output.getvalue().strip()\n    expected_output = \"1\\n2\\n3\\n4\\n5\\n\"\n    assert output == expected_output, f\"Test Case 1 Failed: Expected '{expected_output}', got '{output}'\"\n    print(\"Test Case 1 Passed.\")\n\n    # Test case 2: Edge case - limit is 1\n    print(\"--- Test Case 2: Edge case - limit is 1 ---\")\n    captured_output = io.StringIO()\n    sys.stdout = captured_output\n    simple_counter(1)\n    sys.stdout = sys.__stdout__\n    # Reset redirect\n    output = captured_output.getvalue().strip()\n    expected_output = \"1\\n\"\n    assert output == expected_output, f\"Test Case 2 Failed: Expected '{expected_output}', got '{output}'\"\n    print(\"Test Case 2 Passed.\")\n\n    # Test case 3: Invalid input - zero\n    print(\"--- Test Case 3: Invalid input - zero ---\")\n    captured_output = io.StringIO()\n    sys.stdout = captured_output\n    simple_counter(0)\n    sys.stdout = sys.__stdout__\n    # Reset redirect\n    output = captured_output.getvalue().strip()\n    expected_output = \"Input must be a positive integer.\"\n    assert output == expected_output, f\"Test Case 3 Failed: Expected '{expected_output}', got '{output}'\"\n    print(\"Test Case 3 Passed.\")\n\n    # Test case 4: Invalid input - negative number\n    print(\"--- Test Case 4: Invalid input - negative number ---\")\n    captured_output = io.StringIO()\n    sys.stdout = captured_output\n    simple_counter(-3)\n    sys.stdout = sys.__stdout__\n    # Reset redirect\n    output = captured_output.getvalue().strip()\n    expected_output = \"Input must be a positive integer.\"\n    assert output == expected_output, f\"Test Case 4 Failed: Expected '{expected_output}', got '{output}'\"\n    print(\"Test Case 4 Passed.\")\n\n    # Test case 5: Larger limit\n    print(\"--- Test Case 5: Larger limit ---\")\n    captured_output = io.StringIO()\n    sys.stdout = captured_output\n    simple_counter(10)\n    sys.stdout = sys.__stdout__\n    # Reset redirect\n    output = captured_output.getvalue().strip()\n    expected_output = \"1\\n2\\n3\\n4\\n5\\n6\\n7\\n8\\n9\\n10\\n\"\n    assert output == expected_output, f\"Test Case 5 Failed: Expected '{expected_output}', got '{output}'\"\n    print(\"Test Case 5 Passed.\")\n\n    test_simple_counter()\n}"
```

FIGURE F.3: JSON output

F. EXAMPLES FROM THE DESIGN BASED SYSTEM DEVELOPMENT

Python test cases

```
import io
import sys

def test_simple_counter():
    # Test case 1: Basic functionality
    print("---- Test Case 1: Basic functionality ----")
    captured_output = io.StringIO()
    sys.stdout = captured_output
    simple_counter(5)
    sys.stdout = sys.__stdout__ # Reset redirect
    output = captured_output.getvalue().strip()
    expected_output = "1\n2\n3\n4\n5"
    assert output == expected_output, f"Test Case 1 Failed: Expected '{expected_output}', got '{output}'"
    print("Test Case 1 Passed.")

    # Test case 2: Edge case - limit is 1
    print("---- Test Case 2: Edge case - limit is 1 ----")
    captured_output = io.StringIO()
    sys.stdout = captured_output
    simple_counter(1)
    sys.stdout = sys.__stdout__ # Reset redirect
    output = captured_output.getvalue().strip()
    expected_output = "1"
    assert output == expected_output, f"Test Case 2 Failed: Expected '{expected_output}', got '{output}'"
    print("Test Case 2 Passed.")

    # Test case 3: Invalid input - zero
    print("---- Test Case 3: Invalid input - zero ----")
    captured_output = io.StringIO()
    sys.stdout = captured_output
    simple_counter(0)
    sys.stdout = sys.__stdout__ # Reset redirect
    output = captured_output.getvalue().strip()
    expected_output = "Input must be a positive integer."
    assert output == expected_output, f"Test Case 3 Failed: Expected '{expected_output}', got '{output}'"
    print("Test Case 3 Passed.")

    # Test case 4: Invalid input - negative number
    print("---- Test Case 4: Invalid input - negative number ----")
    captured_output = io.StringIO()
    sys.stdout = captured_output
    simple_counter(-3)
    sys.stdout = sys.__stdout__ # Reset redirect
    output = captured_output.getvalue().strip()
    expected_output = "Input must be a positive integer."
    assert output == expected_output, f"Test Case 4 Failed: Expected '{expected_output}', got '{output}'"
    print("Test Case 4 Passed.")

    # Test case 5: Larger limit
    print("---- Test Case 5: Larger limit ----")
    captured_output = io.StringIO()
    sys.stdout = captured_output
    simple_counter(10)
    sys.stdout = sys.__stdout__ # Reset redirect
    output = captured_output.getvalue().strip()
    expected_output = "1\n2\n3\n4\n5\n6\n7\n8\n9\n10"
    assert output == expected_output, f"Test Case 5 Failed: Expected '{expected_output}', got '{output}'"
    print("Test Case 5 Passed.")

test_simple_counter()
```

FIGURE F.4: Python test cases

Python solution code

```
def simple_counter(limit):
    """
    Counts from 1 up to the given limit and prints each number.

    Args:
        limit (int): The upper bound for the counting.
    """
    if not isinstance(limit, int) or limit < 1:
        print("Input must be a positive integer.")
        return

    for number in range(1, limit + 1):
        print(number)

# Example usage:
simple_counter(5)
```

These are again a lot of programming feature combined into one exercise. Even though this was not asked from the LLM

FIGURE F.5: Limitation: to many features

F.2 Modular generation example

F.2.1 What this iteration enabled

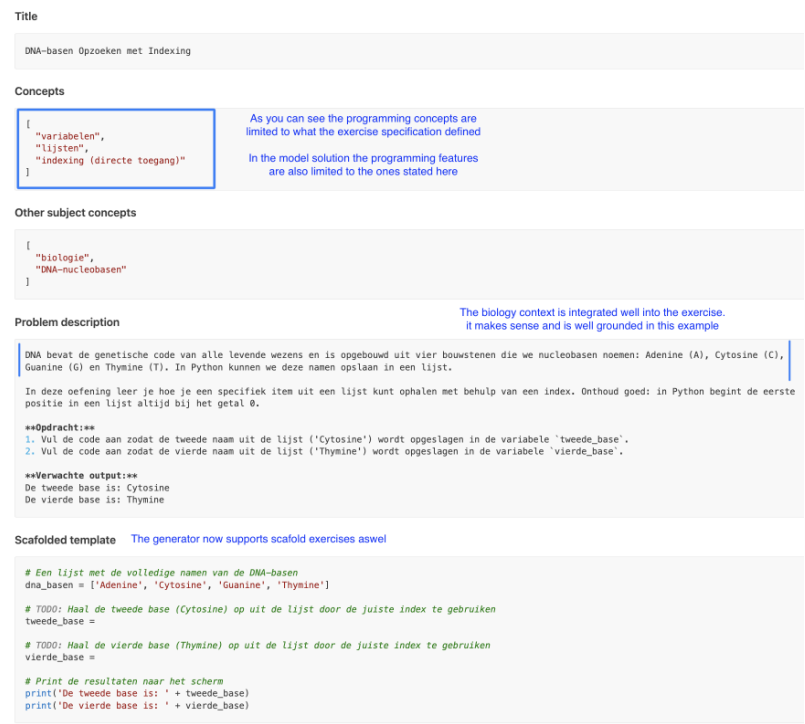


FIGURE F.6: Enter Caption



FIGURE F.7: Enter Caption

F. EXAMPLES FROM THE DESIGN BASED SYSTEM DEVELOPMENT

Title

DNA-letters extraheren uit een string

Concepts

```
[
  "print()",
  "string als lijst behandelen (indexing)",
  "variabelen",
  "strings"
]
```

Other subject concepts

```
[
  "DNA-sequenties",
  "Nucleobasen (A, C, G, T)"
]
```

Problem description

English translation: "In the previous exercise you learned how to get a item from a list using an index"
Here we can see the exercise refers to the previous exercise in the sequence (prior-knowledge mechanism)
see exercise "DNA-base Opzoeken met Indexing"

In de vorige oefening heb je geleerd hoe je met een index een item uit een lijst kunt halen. Wist je dat je in Python een string (tekst) precies hetzelfde kunt behandelen als een lijst? Elke letter in een woord heeft een eigen nummer of index, beginnend bij 0.

In deze oefening gaan we een specifieke letter uit een DNA-code halen. DNA bestaat uit een lange reeks van de letters A, C, G en T.

Gegeven is de DNA-sequentie: ATGCGTGAA.

Opdracht:

1. Vul de code aan zodat de eerste letter van de string 'dna_code' wordt opgeslagen in de variabele 'eerste_letter'. Gebruik hiervoor index 0.
2. Vul de code aan zodat de vijfde letter van de string 'dna_code' wordt opgeslagen in de variabele 'vijfde_letter'. Gebruik hiervoor index 4.

Hint: Denk terug aan hoe je bij de vorige oefening een item uit de lijst 'dna_basen' ophaalde met vierkante haken [].

Verwachte output:
De eerste letter is: A
De vijfde letter is: G

Scaffolded template

```
# Een DNA-sequentie voorgesteld als een string
dna_code = 'ATGCGTGAA'

# TODO: Haal de eerste letter op (index 0) en sla deze op in de variabele
eerste_letter =

# TODO: Haal de vijfde letter op (index 4) en sla deze op in de variabele
vijfde_letter =

# De resultaten worden hier geprint
print('De eerste letter is: ' + eerste_letter)
print('De vijfde letter is: ' + vijfde_letter)
```

FIGURE F.8: Reference to a previous exercise

F.2.2 Current limitations

Title

DNA-letters één voor één printen

Concepts

```
[
  "strings",
  "iteratie (for-loop)",
  "print()"
]
```

Other subject concepts

```
[
  "DNA-sequentie",
  "Stikstofbasen (A, C, T, G)"
]
```

Problem description

The refer to a base as a "letter" this is incorrect in a biology context and can cause confusion

In de vorige oefeningen heb je geleerd hoe je een specifieke letter uit een tekst haalt en hoe je een lijst met namen doorloopt met een for-loop. In Python kun je een for-loop ook direct gebruiken op een string (tekst). De loop gaat dan letter per letter door de tekst heen.

In de biologie is een DNA-sequentie de volgorde van de letters A, C, T en G. Om deze sequentie te analyseren, moeten we vaak elke letter één voor één bekijken. Stel dat we de sequentie 'ACTG' hebben en we willen elke letter apart op het scherm tonen.

Opdracht:

Vul de code aan zodat de for-loop door de string 'dna_sequentie' wandelt. Print voor elke letter (base) de tekst 'Base: ' gevolgd door de letter.

Verwachte output:

Base: A
Base: C
Base: T
Base: G

Hint: Gebruik de kennis uit de vorige oefening over modelorganismen. In plaats van een lijst met namen, gebruik je nu de string 'dna_sequentie'.

FIGURE F.9: Incorrect terminology

F.3. Feedback-Based Refinement Examples

DNA-letters één voor & ↑ ↓ >

Title

De DNA-Basen Vinden!

Concepts

```
[
  "print()",
  "lijsten",
  "Directe toegang tot data in een lijst (Indexing)"
]
```

Other subject concepts

```
[
  "De 4 Nucleobasen van DNA: Adenine, Cytosine, Guanine, Thymine"
]
```

Problem description

English translation: "In the world of DNA there are 4 important bases:....
There are only 4 so this wording could create a misconception that there
exist more then 4 bases

In de wereld van DNA zijn er vier belangrijke basen: Adenine, Cytosine, Guanine en Thymine. Deze basen werken samen om de genetische code te vormen.

We hebben een lijst gemaakt met de volledige namen van deze basen:
'dna_basen' = ["Adenine", "Cytosine", "Guanine", "Thymine"]

Je taak is om, met behulp van de lijst en de kennis van indexen (de posities van elementen in een lijst, beginnend bij 0), de namen van specifieke basen te printen.

==Voorbeeld:==
Als we de naam van de eerste base willen weten, gebruiken we de index '0'. De code zou dan iets zijn als:
'print(dna_basen[0])'
Dit zou 'Adenine' moeten printen.

Kijk naar de code hieronder. Jouw taak is om de 'print()' statements aan te passen zodat de juiste DNA-base wordt getoond.

FIGURE F.10: Wrong phrasing

F.3 Feedback-Based Refinement Examples

F.3.1 What this iteration enabled

Problem description

Here it is said DNA is translated to proteins but this does not happen directly so it is flagged by the feedback agent

In de biologie wordt de informatie in het DNA vertaald naar eiwitten. Dit proces heet translatie. De genetische code wordt gelezen in groepjes van drie letters, die we 'codons' noemen. Een speciaal soort codon is het 'stopcodon', dat aangeeft dat de aanmaak van een eiwit moet stoppen.

In de vorige oefening heb je geleerd hoe je een waarde opzoekt in een dictionary met 'paring(base)'. Nu gaan we dit toepassen op een codontabel. Je krijgt een dictionary genaamd 'codontabel' waarin verschillende codons zijn gekoppeld aan hun functie.

==Opdracht:==
Zoek in de dictionary 'codontabel' op wat de functie is van het codon "TAA" en sla dit op in de variabele 'resultaat'. Print daarna deze variabele.

==Voorbeeld:==
Als je de waarde van "ATG" zou opzoeken, zou de output "start" zijn. Voor "TAA" moet de output exact de tekst hieronder zijn:

Verwachte output:
stop

FIGURE F.11: Teacher agent flags issue

Problem description

Here the mistake flagged by the teacher feedback agent is resolved instead of DNA is translated it now says mRNA is translated to proteins

In de biologie wordt de informatie in het mRNA vertaald naar eiwitten. Dit proces heet translatie. De genetische code wordt gelezen in groepjes van drie letters, die we 'codons' noemen. Een speciaal soort codon is het 'stopcodon', dat aangeeft dat de aanmaak van een eiwit moet stoppen.

In de vorige oefening heb je geleerd hoe je een waarde opzoekt in een dictionary met 'paring(base)'. Nu gaan we dit toepassen op een codontabel. Je krijgt een dictionary genaamd 'codontabel' waarin verschillende codons zijn gekoppeld aan hun functie.

==Opdracht:==
Zoek in de dictionary 'codontabel' op wat de functie is van het codon "UAA" en sla dit op in de variabele 'resultaat'. Print daarna deze variabele.

==Voorbeeld:==
Als je de waarde van "AUG" zou opzoeken, zou de output "start" zijn. Voor "UAA" moet de output exact de tekst hieronder zijn:

Verwachte output:
stop

FIGURE F.12: Teachers flagged issue is successfully resolved

F. EXAMPLES FROM THE DESIGN BASED SYSTEM DEVELOPMENT

Problem description

In de biologie bestaat een DNA-fragment uit een specifieke volgorde van vier stikstofbasen: Adenine (A), Thymine (T), Cytosine (C) en Guanine (G). We noemen deze volgorde een DNA-sequentie.

In de vorige oefeningen heb je twee belangrijke dingen geleerd:

1. Hoe je met een for-loop door een lijst van woorden kunt wandelen.
2. Hoe je losse letters uit een string (tekst) kunt halen met een index.

The student feedback agent flags this, because a list is a data type in python this can wording can possibly cause confusion

In Python kun je een for-loop ook direct op een string gebruiken. De computer behandelt de string dan als een lijst van letters en gaat er één voor één doorheen. Dit is ideaal om een DNA-sequentie base per base te analyseren.

****Jouw opdracht:****
Vul de code aan zodat het programma elke letter (base) uit de variabele 'sequentie' één voor één onder elkaar print.

****Voorbeeld:****
Als de sequentie 'ACT' is, moet de output zijn:
A
C
T

****Verwachte output voor deze oefening:****
G
A
T
C
G

FIGURE F.13: Student agent flags issue

Problem description

In de biologie bestaat een DNA-fragment uit een specifieke volgorde van vier stikstofbasen: Adenine (A), Thymine (T), Cytosine (C) en Guanine (G). We noemen deze volgorde een DNA-sequentie.

In de vorige oefeningen heb je twee belangrijke dingen geleerd:

1. Hoe je met een for-loop door een lijst van woorden kunt wandelen.
2. Hoe je losse letters uit een string (tekst) kunt halen met een index.

So the problem flagged by the student feedback agent is resolved as follows:
Before: "A computer treats a string as a list of letters..."
After: "a computer goes char by char through the string..."

In Python kun je een for-loop ook direct op een string gebruiken. De computer loopt dan karakter per karakter door de string heen. Dit is ideaal om een DNA-sequentie base per base te analyseren.

****Jouw opdracht:****
Vul de code aan zodat het programma elke letter (base) uit de variabele 'sequentie' één voor één onder elkaar print.

****Voorbeeld:****
Als de sequentie 'ACT' is, moet de output zijn:
A
C
T

****Verwachte output voor deze oefening:****
G
A
T
C
G

FIGURE F.14: Students flagged issue is successfully resolved

F.3.2 Current limitation

Python solution code

```
dna_sequentie = 'ATCGATCGATATGCGTACG'
fragment = dna_sequentie[10:15]
print(fragment)
```

Python test cases

```
import io
from contextlib import redirect_stdout

def test_exercise():
    # Setup test input
    test_dna = 'ATCGATCGATATGCGTACG'
    expected_val = 'ATGCG'

    # Capture output of the logic
    f = io.StringIO()
    with redirect_stdout(f):
        try:
            # We repeat the student logic internally for validation
            # This simulates running the code defined in solution
            fragment_result = test_dna[10:15]
            print(fragment_result)
        except Exception as e:
            assert False, f"Er is een fout in de code opgetreden: {e}"

    output = f.getvalue().strip().split('\n')

    # Check 1: Is there any output?
    if not output or output[0] == '':
        assert False, "Fout: Je hebt niets geprint. Gebruik de print() functie."

    # Check 2: Too many lines?
    if len(output) > 1:
        assert False, "Fout: Je hebt te veel regels geprint. Print alleen het fragment."

    # Check 3: Is the value correct?
    actual_val = output[0]
    if actual_val != expected_val:
        if actual_val == 'ATGCGT':
            assert False, f"Fout: Je hebt index 15 ook meegeteld. Gebruik [10:15] in plaats van [10:16]."
```

The test case just test its own implementation of the problem:
1)executing this test and model solution would not be a valid check for the model solution
2) if the student implements a wrong solution. The test cases will not catch it because it executes its own implementation

```
        elif actual_val == 'TCGAT':
            assert False, f"Fout: Je bent op de verkeerde index begonnen. Python begint te tellen bij 0."
        else:
            assert False, f"Fout: De output was {actual_val}, maar we verwachten {expected_val}. Controleer je slicing indices (start:stop)."
```

```
test_exercise()
```

FIGURE F.15: Problem in test case generation

F.4 Dedicated Test-Generation Example

F.4.1 What this iteration enabled

Python solution code

```
dna_sequentie = 'ATCGATCGATATCGGTACG'
fragment = dna_sequentie[10:15]
print(fragment)
```

Python test cases

```
import io
from contextlib import redirect_stdout

def test_exercise():
    # Setup test input
    test_dna = 'ATCGATCGATATCGGTACG'
    expected_val = 'ATCGG'

    # Capture output of the logic
    f = io.StringIO()
    with redirect_stdout(f):
        try:
            # We repeat the student logic internally for validation
            # This simulates running the code defined in solution
            fragment_result = test_dna[10:15]
            print(fragment_result)
        except Exception as e:
            assert False, f"Er is een fout in de code opgetreden: {e}"

    output = f.getvalue().strip().split('\n')

    # Check 1: Is there any output?
    if not output or output[0] == '':
        assert False, "Fout: Je hebt niets geprint. Gebruik de print() functie."

    # Check 2: Too many lines?
    if len(output) > 1:
        assert False, "Fout: Je hebt te veel regels geprint. Print alleen het fragment."

    # Check 3: Is the value correct?
    actual_val = output[0]
    if actual_val != expected_val:
        if actual_val == 'ATCGGT':
            assert False, f"Fout: Je hebt index 15 ook meegeteld. Gebruik [10:15] in plaats van [10:16]."
        elif actual_val == 'TCGAT':
            assert False, f"Fout: Je bent op de verkeerde index begonnen. Python begint te tellen bij 0."
        else:
            assert False, f"Fout: De output was {actual_val}, maar we verwachten {expected_val}. Controleer je slicing indices (start:stop)."

test_exercise()
```

The test case just test its own implementation of the problem:
 1) executing this test and model solution would not be a valid check for the model solution
 2) if the student implements a wrong solution. The test cases will not catch it because it executes its own implementation

FIGURE F.16: Identified test case issue

Test Cases

```
import sys
import io
from contextlib import redirect_stdout

# Helper-functie om de studentencode uit te voeren en de output op te vangen
def run_test():
    try:
        with redirect_stdout(f):
            # Voer de studentcode uit. De variabele 'student_code' wordt door het platform aangeleverd.
            exec(student_code, {})
    except IndexError:
        assert False, 'Fout: Je probeert een index te gebruiken die niet bestaat. Onthoud dat een lijst met 4 items indexen 0 tot en met 3 heeft.'
    except Exception as e:
        assert False, f'Er is een fout opgetreden bij het uitvoeren van je code: {e}'

    # Haal de geprinte regels op, verwijder witruimte random
    output = f.getvalue().splitlines()
    output = [line.strip() for line in output if line.strip()]

    # Controleer of er iets is geprint
    assert len(output) > 0, 'Je hebt nog niets geprint. Gebruik de print() functie om de resultaten te tonen.'

    # Controleer op het juiste aantal regels
    assert len(output) == 2, f'Je hebt {len(output)} regels geprint, maar we verwachten er precies 2.'

    # Verwachte waarden
    expected_1 = 'De tweede base is Cytosine'
    expected_2 = 'De vierde base is Thymine'

    # Regel 1 validatie (De tweede base - Cytosine)
    line1 = output[0]
    if 'Adenine' in line1:
        assert False, 'Fout op regel 1: Je hebt de eerste base (index 0) geprint in plaats van de tweede. In Python begin je te tellen bij 0.'
    if 'Guanine' in line1:
        assert False, 'Fout op regel 1: Je hebt de derde base (index 2) geprint in plaats van de tweede.'
    assert line1 == expected_1, f'Regel 1 klopt niet helemaal. \nVerwacht: "{expected_1}" \nJij printte: "{line1}"'

    # Regel 2 validatie (De vierde base - Thymine)
    line2 = output[1]
    if 'Guanine' in line2:
        assert False, 'Fout op regel 2: Je hebt de derde base (index 2) geprint in plaats van de vierde.'
    if 'Adenine' in line2:
        assert False, 'Fout op regel 2: Je hebt de eerste base (index 0) geprint in plaats van de vierde.'
    assert line2 == expected_2, f'Regel 2 klopt niet helemaal. \nVerwacht: "{expected_2}" \nJij printte: "{line2}"'

    # Start de validatie
    run_test()
```

Here we can see the test now executes the code in the variable student_code.
 As long as the execution environment puts the model solution in that variable the test are correctly run

FIGURE F.17: How this iteration solved that issue

F.5. eTeacher Y ML output and platform specific test cases

Test Cases

```
import io
from contextlib import redirect_stdout

# Voorbereiden van de test
f = io.StringIO()

try:
    with redirect_stdout(f):
        # Voer de studentcode uit
        exec(student_code)
except Exception as e:
    assert False, f'Je code bevat een fout en kon niet worden uitgevoerd: {e}'

# Verwerk de geprinte output
# .strip() verwijdert witruimte aan het begin/einde van de gehele output
# .split('\n') splitst de output in een lijst van regels
output_lines = [
    line.strip()
    for line in f.getvalue().strip().split('\n')
    if line.strip()
]

# Foutdetectie: Geen output
assert len(output_lines) > 0, (
    'Je hebt niets geprint. Gebruik de print() functie om het resultaat te tonen.'
)

# Foutdetectie: Verkeerd aantal regels
assert len(output_lines) == 2, (
    f'Je hebt {len(output_lines)} regels tekst geprint, '
    'maar we verwachten er precies 2.'
)

# Foutdetectie: Controleer de inhoud van de eerste regel
expected_1 = 'De eerste base is A'
actual_1 = output_lines[0]

assert actual_1 == expected_1, (
    f'De eerste regel is niet correct.\n'
    f'Verwacht: "{expected_1}"\n'
    f'Jouw output: "{actual_1}"\n'
    'Tip: Gebruik codon[0] om de eerste letter op te halen.'
)

# Foutdetectie: Controleer de inhoud van de tweede regel
expected_2 = 'De derde base is G'
actual_2 = output_lines[1]

assert actual_2 == expected_2, (
    f'De tweede regel is niet correct.\n'
    f'Verwacht: "{expected_2}"\n'
    f'Jouw output: "{actual_2}"\n'
    'Tip: Gebruik codon[2] om de derde letter op te halen. '
    'Onthoud dat Python begint met tellen bij 0!'
)
```

This shows that the test generation can generate test cases that provide meaningful feedback

FIGURE F.18: Generates meaningful feedback when test fails

F.5 eTeacher Y ML output and platform specific test cases

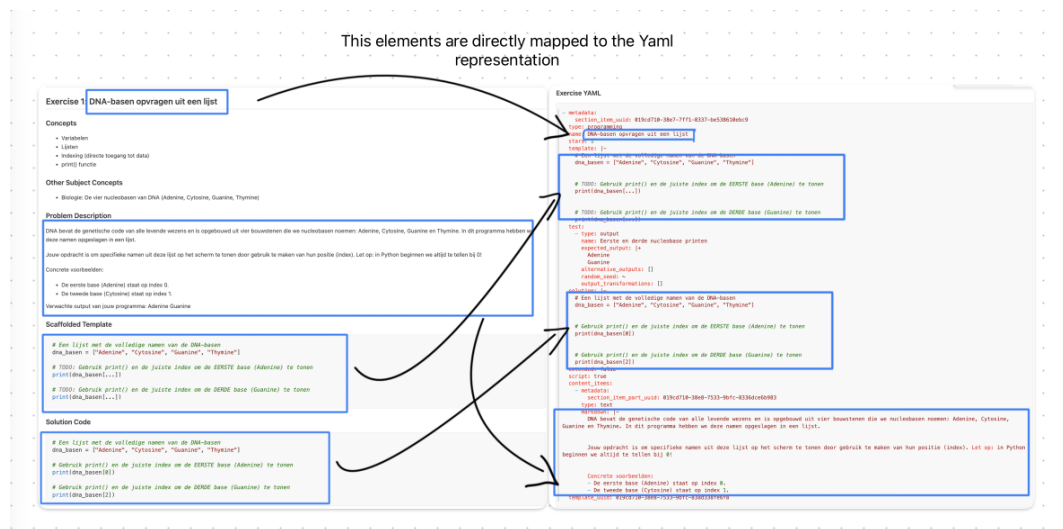


FIGURE F.19: Direct mapping to eTeacher format



FIGURE F.20: Successful test generation but bad feedback generation

F.6 External review notes

“latex

F.6.1 First external review

Initial feedback

The reviewer could not edit the PDF directly but suggested the following changes:

- **Exercise 5:** Change the example to:

Voorbeeld: ls dna_sequentie1 = ' TGC' en dna_sequentie2 = ' TCG', dan is de output 2. De nucleotiden op positie 0 () en positie 1 (T) zijn hetzelfde. Op positie 2 staat in de eerste sequentie een G, versus een C in de tweede sequentie, en op positie 3 staat C in de eerste sequentie versus G in de tweede sequentie. Deze nucleotiden komen dus niet overeen.

De output wordt hierbij veranderd van 3 naar 2 en ook het niet overeenkomen van de basen op positie 2 wordt vermeld.

- **Exercise 4:** In de lijst met modelorganismen staat *Drosophila*, terwijl dit de naam van het geslacht is. Dit kan beter worden vervangen door *Drosophila melanogaster* of *D. melanogaster*, aangezien dit het organisme aanduidt.
- **Exercise 1:** De formulering

We hebben de volledige naam van vier belangrijke basen in een lijst gezet.

kan suggereren dat er meer dan vier basen zijn. Dit zou mogelijk een misconceptie kunnen introduceren. De voorgestelde formulering is:

We hebben de volledige benamingen van de vier basen waaruit DN is opgebouwd in een lijst gezet.

Feedback per exercise

Exercise 1 DN is opgebouwd uit nucleotiden. Dit zijn suikers met daarop een fosfaatgroep en een van de basen. In plaats van de eerste zin

DN is een informatiedrager die is opgebouwd uit vier verschillende bouwstenen, de nucleobasen.

wordt de volgende formulering voorgesteld:

DN is een informatiedrager die is opgebouwd uit vier verschillende nucleotiden. Dit zijn suikers met daarop vier verschillende nucleobasen.

Exercise 2 Check, in orde.

Exercise 3 Check, in orde.

Exercise 4 Check, in orde.

Exercise 5 Check, in orde.

Exercise 6 Check, in orde.

General comment Lange programmateksten zijn niet volledig zichtbaar in de PDF.

Exercise 7 Check, in orde.

Exercise 8 Check, in orde.

Exercise 9 Check, in orde.

Is mogelijke aanvullende achtergrond of motivatie kan het GC-gehalte worden vermeld. Dit behoort echter niet tot de leerstof van het secundair onderwijs. Een GC-basepaar is sterker dan een T-basepaar. Daarom kan het bepalen van het GC-gehalte in bepaalde situaties belangrijk zijn.

Exercise 10 fwezig.

Exercise 11 Check, in orde.

Exercise 12 Check, in orde.

Exercise 13 De `start_index` kan mogelijk beter verwijzen naar het begin van het startcodon (`TG`). Deze index zou in het oorspronkelijke voorbeeld dan 0 zijn in plaats van 2.

Het is daarnaast beter om het startcodon in de sequentie van de oefening op te nemen, overeenkomstig de opmerking bij Exercise 14. Om de oefening biologisch volledig correct te maken, kan ook een stopcodon worden toegevoegd. Dit stopcodon moet zich in het juiste leesraam bevinden. Het stopcodon kan eventueel worden weggelaten, maar het gebruik van een startcodon wordt wel aanbevolen.

Een mogelijk voorbeeld is:

```
Sequentie:   CCC TGGCTG  T GGG
start_index: 3
gen_lengte:  9
```

De `start_index` is hier 3, omdat het startcodon `TG` op die positie begint.

Exercise 14 Translatie start steeds met het startcodon `TG`. Het is daarom beter om dit codon in de sequentie te plaatsen en de leerlingen het startcodon te laten zoeken. Dit sluit ook aan bij de volgende oefening, waarin de stopcodons expliciet worden benoemd.

Exercise 15 Check, in orde.

General comment on exercise order Het kan logischer zijn om Exercises 14 en 15 vóór Exercises 12 en 13 te plaatsen.

Exercise 16 De volledige namen van de nucleobasen kunnen het beste eerst worden vermeld, gevolgd door de afkortingen tussen haakjes:

DN bestaat uit vier bouwstenen die we nucleobasen noemen: `denine` (), Thymine (T), Cytosine (C) en Guanine (G).

Exercise 17 Om misconcepties te vermijden, kan het woord “letters” worden vervangen door “nucleobasen”:

In de biologie wordt de genetische code gelezen in groepjes van drie nucleobasen, die we mRNA-codons noemen.

Exercise 18 Het is duidelijker om te spreken over een eiwit of een aminozuurketen. De term “eiwitketen” wordt doorgaans niet gebruikt. Een of meerdere ketens van aminozuren vormen samen een eiwit.

Exercise 19 Goed.

Exercise 20 In orde.

Exercise 21 Een biologisch correctere formulering is:

Een eiwit bestaat uit een of meerdere ketens van aminozuren. Deze vertaling gebeurt in stapjes van drie nucleobasen, die we codons noemen.

Earlier feedback

Exercise 1 Er zijn slechts vier basen in DN . De volgende formulering kan daarom misleidend zijn:

We hebben de volledige naam van vier belangrijke DN -basen in een lijst gezet.

Exercise 3 De namen van de modelorganismen moeten op een consistente manier worden geschreven, bijvoorbeeld:

E. coli, *C. elegans*, *M. musculus*, *D. melanogaster*, ...

Exercise 5 Het voorbeeld bevat een fout. De nucleotiden op posities 0 en 1 zijn hetzelfde, terwijl de nucleotiden op posities 2 en 3 verschillend zijn. Een mogelijke oplossing is om alleen positie 3 verschillend te maken, bijvoorbeeld door de tweede sequentie te veranderen naar TGG.

F.6.2 Second external review

Feedback per exercise

Exercise 4 Indien mogelijk kan de term “stikstofbasen” worden vervangen door “nucleobasen”. Hierdoor wordt dezelfde terminologie gebruikt als in Exercise 1.

Exercise 6 De formulering

bijvoorbeeld om het 3'-einde van de streng te bestuderen

is niet volledig correct. De volgende formulering wordt voorgesteld:

bijvoorbeeld om de templatestreng in de juiste richting te bestuderen

Exercise 10 De term “codon” kan alleen worden gebruikt voor een RN -sequentie. Bij een DN -sequentie moet de term “triplet” worden gebruikt.

Exercise 14 In deze oefening wordt de term “triplet” gebruikt, maar aan het begin van de oefening komt ook nog de term “codon” voor.

Exercise 15 In deze oefening wordt opnieuw alleen de term “codon” gebruikt. Omdat het om een DN -sequentie gaat, moet hier “triplet” worden gebruikt.

Exercise 17 De term “codon” wordt hier correct gebruikt.

Exercise 20 De term “codon” wordt hier incorrect gebruikt.

Exercise 21 De term “codon” wordt hier correct gebruikt. ““

Appendix G

Detailed Consistency Evaluation Results

G.1 Teacher Blueprints

TABLE G.1: Pairwise correlations between consistency metrics for teacher-defined exercise Blueprints.

	Sol. CB	Test CB	Temp. CB	Desc. BLEU	Desc. Sim.	Emb. Sim.	Met. J c.
Sol. CB	1.000	0.063	0.683	0.265	0.059	0.086	-0.025
Test CB	0.063	1.000	0.009	-0.058	-0.130	-0.146	0.108
Temp. CB	0.683	0.009	1.000	0.195	0.045	0.042	-0.149
Desc. BLEU	0.265	-0.058	0.195	1.000	0.564	0.469	-0.092
Desc. Sim.	0.059	-0.130	0.045	0.564	1.000	0.311	-0.083
Emb. Sim.	0.086	-0.146	0.042	0.469	0.311	1.000	0.031
Met. J c.	-0.025	0.108	-0.149	-0.092	-0.083	0.031	1.000

Not. CB = CodeBLEU; Desc. = Description; Sim. = Similarity; Emb. = Embedding; Met. J c. = Met. J c. = Met. J c. =

TABLE G.2: Mean pairwise similarity scores per teacher-defined exercise blueprint across 30 generation runs.

Exercise	Runs	Sol. CB	Test CB	Temp. CB	BLEU	Text Sim.	Emb. Sim.	Met. J c.
lueprint_1.md	30	0.379	0.131	0.397	0.235	0.161	0.874	0.568
lueprint_2.md	30	0.107	0.128	0.130	0.227	0.158	0.855	0.605
lueprint_3.md	30	0.587	0.148	0.487	0.293	0.166	0.843	0.603
lueprint_4.md	30	0.420	0.133	0.294	0.321	0.216	0.906	0.772
lueprint_5.md	30	0.426	0.084	0.387	0.353	0.285	0.899	0.470

Not. CB = CodeBLEU; Sol. = Solution; Temp. = Template; Sim. = Similarity; Emb. = Embedding; Met. J c. = Met. J c. = Met. J c. =

G.2 Generated blueprint

G. DETAILED CONSISTENCY EVALUATION RESULTS

TABLE G.3: Mean pairwise similarity scores per system-generated exercise blueprint across 30 generation runs.

Exercise	Runs	Sol. CB	Test CB	Temp. CB	BLEU	Text Sim.	Emb. Sim.	Met . J c.
spec_1.md	30	0.257	0.067	0.318	0.192	0.164	0.868	0.776
spec_2.md	30	0.379	0.048	0.335	0.260	0.178	0.845	0.738
spec_3.md	30	0.535	0.069	0.386	0.316	0.193	0.886	0.718
spec_4.md	30	0.549	0.079	0.366	0.328	0.172	0.895	0.512
spec_5.md	30	0.676	0.066	0.454	0.297	0.179	0.865	0.477

Not . CB = CodeBLEU; Sol. = Solution; Temp. = Template; Sim. = Similarity; Emb. = Embedding; Met . J c. = Met d t J cc rd.

TABLE G.4: Pairwise correlations between consistency metrics for system-generated exercise blueprint.

	Sol. CB	Test CB	Temp. CB	Desc. BLEU	Desc. Sim.	Emb. Sim.	Met . J c.
Sol. CB	1.000	0.190	0.558	0.376	0.083	0.206	-0.267
Test CB	0.190	1.000	0.083	0.138	-0.010	0.184	-0.066
Temp. CB	0.558	0.083	1.000	0.231	0.127	0.119	-0.080
Desc. BLEU	0.376	0.138	0.231	1.000	0.383	0.429	-0.183
Desc. Sim.	0.083	-0.010	0.127	0.383	1.000	0.200	0.005
Emb. Sim.	0.206	0.184	0.119	0.429	0.200	1.000	-0.056
Met . J c.	-0.267	-0.066	-0.080	-0.183	0.005	-0.056	1.000

Not . CB = CodeBLEU; Desc. = Description; Sim. = Similarity; Emb. = Embedding; Met . J c. = Met d t J cc rd.

ppendix H

Feedback- gent Evaluation

This appendix presents representative examples from the qualitative evaluation of the teacher and student feedback agents.

H.1 Quantitative Results

Te cher-defined specs (T1-T5)										
Spec	Exercises refined,	Exercises refined, %	Te cher	gent only	Student	gent only	Both	gents	Solution ch nged	Description ch nged
T1	7	23.3%		2		3	2		1	6
T2	10	33.3%		8		1	1		6	9
T3	16	53.3%		15		0	1		1	9
T4	6	20.0%		3		1	2		1	6
T5	18	60.0%		5		4	9		5	10
Tot l (te cher-defined)	57	38.0%		33		9	15		14	40
System-gener ted specs (G1-G5)										
G1	18	60.0%		4		4	10		3	16
G2	13	43.3%		5		2	6		4	12
G3	13	43.3%		5		4	4		4	12
G4	18	60.0%		8		5	5		6	17
G5	19	63.3%		9		5	5		2	19
Tot l (system-gener ted)	81	54.0%		31		20	30		19	76

TABLE H.1: Feedback-agent verdicts and component changes for teacher-defined and system-generated specifications. Each specification was processed 30 times. “Teacher agent only,” “student agent only,” and “both agents” indicate which feedback perspectives requested revision. Component-change counts indicate runs in which the draft and final component were not identical. (abbreviations: T = teacher-defined specification; G = system-generated specification.)

Teacher-defined specs (T1-T5)																
Spec	Solution	CodeBLEU	Test c	ses	CodeBLEU	Templ	te	CodeBLEU	BLEU	Text sim.	Embedding sim.	Met d	t	J	cc	rd
T1		0.379			0.131			0.397	0.235	0.161	0.874					0.568
T2		0.107			0.128			0.130	0.227	0.158	0.855					0.605
T3		0.587			0.148			0.487	0.293	0.166	0.843					0.603
T4		0.420			0.133			0.294	0.321	0.216	0.906					0.772
T5		0.426			0.084			0.387	0.353	0.285	0.899					0.470
System-generated specs (G1-G5)																
G1		0.257			0.067			0.318	0.192	0.164	0.868					0.776
G2		0.379			0.048			0.335	0.260	0.178	0.845					0.738
G3		0.535			0.069			0.386	0.316	0.193	0.886					0.718
G4		0.549			0.079			0.366	0.328	0.172	0.895					0.512
G5		0.676			0.066			0.454	0.297	0.179	0.865					0.477

TABLE H.2: Final similarity metrics for teacher-defined and system-generated specifications (T = teacher-defined specification; G = system-generated specification). Values are reported per specification.

H.2 Representative Feedback- gent Refinements

H.2.1 alignment between exercise components

- **FeedbackTypes.TEACHER**
 - **Refinement Prompt:** Rewrite the exercise by applying the summary and all suggested fixes exactly. Ensure the example input string is 'AGTCCG' and the expected output consists of the letters A, G, T, C, C, G each on a new line. Do not change the difficulty, the use of the while-loop, or the hardcoded limit of 6 in the template. 1
 - **Issue:** The problem description mentions the string 'AGTCCG' (length 6) and the template hardcodes the loop to run 6 times, but the example input provided is 'GATC' (length 4) with its corresponding output. 2
 - **Suggestion:** Change the example input to 'AGTCCG' and update the expected output to show the characters A, G, T, C, C, G each on a new line to maintain consistency with the instructions and the code template.
- **FeedbackTypes.STUDENT**
 - **Refinement Prompt:** Rewrite the exercise to ensure the DNA string 'AGTCCG' is used consistently across the problem description, the example input/output, and the scaffolded template. Update the expected output to show the letters A, G, T, C, C, G on separate lines. 2
 - **Issue:** The problem description says the user will work with 'AGTCCG', but the 'Voorbeeld van invoer' uses 'GATC'. Because the template hardcodes 'while teller < 6', using the example string 'GATC' would result in an `IndexError`. 2
 - **Suggestion:** Update the 'Voorbeeld van invoer' to use `dna_sequentie = 'AGTCCG'` and update the 'Verwachte uitvoer' to show the letters A, G, T, C, C, and G on separate lines. [1,2]
- **FeedbackTypes.MERGE**
 - **Refinement Prompt:** Rewrite the exercise to ensure the DNA string 'AGTCCG' is used consistently across the problem description, example input/output, and the scaffolded template. Update the example input to 'AGTCCG' and the expected output to show the letters A, G, T, C, C, G on separate lines. Do not change the difficulty, the while-loop structure, or the hardcoded limit of 6 in the template. [1,2]
 - **Issue:** The problem description and code template use 'AGTCCG' (length 6), but the example input uses 'GATC' (length 4). This inconsistency would cause an `IndexError` if a student runs the template code with the example input. [1,2]
 - **Suggestion:** Change the example input to 'AGTCCG' and update the expected output to show the characters A, G, T, C, C, and G on separate lines. [1,2]

FIGURE H.1: inconsistency was flagged by both the students and the teacher agent

```

--- Draft Description
+++ Final Description
@@ -3,10 +3,12 @@
Je krijgt een DNA-string "AGTCCG". Je moet een while-loop gebruiken
en een teller-variabele om elke positie in de string te bezoeken. De
teller begint bij 0 en gaat tot aan de lengte van de string.

Voorbeeld van invoer:
-dna_sequentie = "GATC"
+dna_sequentie = "AGTCCG"
The flagged issue is solved as follows the example is changed by changing the example
Verwachte uitvoer:
+A
G
-A
T
C
+C
+G

```

FIGURE H.2: Enter Caption

H.2.2 Scientific and domain-specific terminology

Applied Feedback Agents

• FeedbackTypes.TEACHER

- **Refinement Prompt:** Rewrite the exercise by replacing the term 'letters' with 'nucleotiden' (nucleotides) and 'DNA-letters' with 'DNA-nucleotiden' in the title and description. Change 'deze code' to 'deze sequentie' when referring to the string value. Ensure all comments in the template and solution code also reflect this change (e.g., 'eerste nucleotide' instead of 'eerste letter'). Keep all Python logic and indexing exactly the same. An issue is flagged for not using the right terminology
- **Issue:** The term 'letters' is used to describe the nitrogenous bases/nucleotides in DNA, which is scientifically informal. Additionally, using 'code' to refer to the variable's value can be ambiguous in a programming context.
- **Suggestion:** Replace 'letters' with 'nucleotiden' and 'deze code' with 'deze sequentie' in the problem description and comments.

FIGURE H.3: Not the correct terminology used

Template Diff (Draft -> Final)

```

--- Draft Template
+++ Final Template
@@ -1,10 +1,10 @@
dna_sequentie = 'GCTAGA'

-# TODO: Gebruik de juiste index om de 1e letter te printen
+# TODO: Gebruik de juiste index om de 1e base te printen
print(dna_sequentie[...])

-# TODO: Gebruik de juiste index om de 3e letter te printen
+# TODO: Gebruik de juiste index om de 3e base te printen
print(dna_sequentie[...])

-# TODO: Gebruik de juiste index om de 6e letter te printen
+# TODO: Gebruik de juiste index om de 6e base te printen
print(dna_sequentie[...])

```

Solution Code Diff (Draft -> Final)

```

--- Draft Solution
+++ Final Solution
@@ -1,10 +1,10 @@
dna_sequentie = 'GCTAGA'

-# De 1e letter staat op index 0
+# De 1e base staat op index 0
print(dna_sequentie[0])

-# De 3e letter staat op index 2
+# De 3e base staat op index 2
print(dna_sequentie[2])

-# De 6e letter staat op index 5
+# De 6e base staat op index 5
print(dna_sequentie[5])

```

FIGURE H.4: Correcting terminology mistake in template and solution code

Description Diff (Draft -> Final)

```

--- Draft Description
+++ Final Description
@@ -1,16 +1,16 @@
-In de vorige oefening heb je geleerd hoe je elementen uit een lijst kunt halen met een index-getal
tussen vierkante haken [ ]. Een string (tekst) in Python kun je op precies dezelfde manier
behandelen. Je kunt een string zien als een geordende lijst van letters.
+In de vorige oefening heb je geleerd hoe je elementen uit een lijst kunt halen met een index-getal
tussen vierkante haken [ ]. Een string (tekst) in Python kun je op precies dezelfde manier
behandelen. Je kunt een string zien als een geordende lijst van basen (de letters in de string staan
voor de stikstofbasen van het DNA).

Bijvoorbeeld: als we de string 'ACT' hebben, dan is:
-- index 0 de letter 'A'
-- index 1 de letter 'C'
-- index 2 de letter 'T'
+- index 0 de base 'A'
+- index 1 de base 'C'
+- index 2 de base 'T'

-In deze oefening hebben we een korte DNA-sequentie: 'GCTAGA'. Jouw taak is om specifieke letters
uit deze tekst te halen en te printen.
+In deze oefening hebben we een korte DNA-sequentie: 'GCTAGA'. Jouw taak is om specifieke basen uit
deze tekst te halen en te printen.

Opdracht:
-1. Print de 1e letter van de sequentie.
-2. Print de 3e letter van de sequentie.
-3. Print de 6e (laatste) letter van de sequentie.
+1. Print de 1e base van de sequentie.
+2. Print de 3e base van de sequentie.
+3. Print de 6e (laatste) base van de sequentie.

Verwachte uitvoer:
G

```

FIGURE H.5: Correcting terminology mistake in problem description

H.2.3 alignment between instructions and scaffold

Applied Feedback Agents

Alignment issue flagged between provided scaffold and problem description

- FeedbackTypes: STUDENT
 - Refinement Prompt: Rewrite the 'Opdracht' section to align with the provided template. Remove the redundant first step about creating the while-loop. Ensure the remaining steps clearly instruct the student to use the 'base' variable and update 'aantal_adenine' within the existing loop structure. Keep the tone appropriate for a secondary-school beginner.
 - Issue: Instruction 1 ('Gebruik een while-loop...') is already present in the scaffolded template, which might make students think they need to rewrite or modify the loop header.
 - Suggestion: Remove the first instruction or rephrase it to explain the purpose of the existing loop rather than presenting it as a task.
 - Issue: Instruction 2 mentions 'de base op de huidige positie', which is slightly redundant since the variable 'base' is already extracted in the template.
 - Suggestion: Simplify instruction 2 to: 'Gebruik binnen de loop een if-statement om te controleren of de variabele base gelijk is aan 'A'.'
 - Issue: Instruction 3 is slightly verbose by repeating the technical term 'accumulator-variabele'.
 - Suggestion: Change instruction 3 to: 'Als de base een 'A' is, verhoog dan de variabele 'aantal_adenine' met 1.'

FIGURE H.6: Instruction in the problem statement and scaffold are not aligned

Description Diff (Draft -> Final)

```

--- Draft Description
+++ Final Description
@@ -3,9 +3,8 @@
@@ -3,9 +3,8 @@
Tijdens het doorlopen van de string moet je ook tellen hoe vaak de base 'A' (Adenine) voorkomt.
Hiervoor gebruik je een 'accumulator-variabele' (een variabele die je telkens met 1 verhoogt als je
een 'A' vindt).

Opdracht:
-1. Gebruik een while-loop die blijft draaien zolang de teller kleiner is dan de limiet.
-2. Gebruik binnen de loop een if-statement om te controleren of de base op de huidige positie
gelijk is aan 'A'.
-3. Als de base een 'A' is, verhoog dan de accumulator-variabele 'aantal_adenine' met 1.
+1. Gebruik binnen de loop een if-statement om te controleren of de variabele base gelijk is aan
'A'.
+2. Als de base een 'A' is, verhoog dan de variabele 'aantal_adenine' met 1.

Voorbeeld:
Als de dna_sequentie 'AGGAT' is en de limiet is 3, dan kijkt het programma alleen naar de eerste
drie letters ('AGG'). Omdat daar één 'A' in zit, is de output:

```

FIGURE H.7: The instruction in the problem statement were adjusted

H.2.4 Clarification of ambiguous programming instructions

Applied Feedback Agents

Issue flagged: Ambiguous wording could lead to wrong understanding of what should be done

- **FeedbackTypes.TEACHER**
 - **Refinement Prompt:** Rewrite the exercise. In the problem_description, replace 'maar telkens 3 posities overslaat' with 'waarbij de teller telkens met 3 wordt verhoogd'. Ensure all other biological and programming logic remains the same, maintaining the example of indices 0, 3, and 6.
 - **Issue:** The phrase 'telkens 3 posities overslaat' is ambiguous. In programming, skipping 3 elements usually implies landing on the 4th, whereas the goal is to increment the index by 3 (moving to the start of the next triplet).
 - **Suggestion:** Change 'telkens 3 posities overslaat' to 'stappen van 3 basen neemt' or 'de teller telkens met 3 verhoogt'.
- **FeedbackTypes.STUDENT**
 - **Refinement Prompt:** Rewrite the exercise by replacing the phrase 'telkens 3 posities overslaat' with 'telkens stappen van 3 neemt' to ensure the student understands the increment should be 3, maintaining consistency with the biological concept of codons and the provided code comments.
 - **Issue:** De tekst gebruikt 'telkens 3 posities overslaat', wat tegenstrijdig kan zijn met de eerdere uitleg '3 posities verder springen' en kan leiden tot een foutieve increment van 4.
 - **Suggestion:** Vervang 'telkens 3 posities overslaat' door 'telkens stappen van 3 neemt'.
- **FeedbackTypes.MERGE**
 - **Refinement Prompt:** Rewrite the exercise. In the problem_description, replace 'telkens 3 posities overslaat' with 'waarbij de teller telkens met 3 wordt verhoogd' to clarify that the loop index should be incremented by 3 in each iteration, maintaining the example of indices 0, 3, and 6.
 - **Issue:** The phrase 'telkens 3 posities overslaat' is ambiguous in a programming context. It could be interpreted as skipping three elements and landing on the fourth (index + 4), whereas the intended logic is to increment the index by 3.
 - **Suggestion:** Replace 'telkens 3 posities overslaat' with 'waarbij de teller telkens met 3 wordt verhoogd'.

FIGURE H.8: This wording could lead to confusion

Description Diff (Draft -> Final)

```

--- Draft Description
+++ Final Description
@@ -3,7 +3,7 @@
In de biologie wordt DNA echter vaak gelezen in groepjes van drie basen, die we 'codons' noemen. Om
de start van elk nieuw codon te vinden, moeten we niet 1 stapje zetten, maar steeds 3 posities
verder springen in de reeks.

Opdracht:
-Vul de code aan zodat het programma de DNA-sequentie doorloopt, maar telkens 3 posities overslaat.
+Vul de code aan zodat het programma de DNA-sequentie doorloopt, waarbij de teller telkens met 3
In elke stap print je de base die op de startpositie van het huidige codon staat. Gebruik de
variabele 'limiet' om te bepalen tot waar de loop moet lopen.
-Vul de code aan zodat het programma de DNA-sequentie doorloopt, waarbij de teller telkens met 3
+Vul de code aan zodat het programma de DNA-sequentie doorloopt, waarbij de teller telkens met 3
wordt verhoogd. In elke stap print je de base die op de startpositie van het huidige codon staat.
Gebruik de variabele 'limiet' om te bepalen tot waar de loop moet lopen.

Voorbeeld:
Als dna_sequentie = 'GCTATAGG' en limiet = 9:

```

FIGURE H.9: The wording was improved

ppendix I

Test Execution and Refinement Behavior

This appendix provides supplementary material for the evaluation of test-case generation and execution-based refinement. It contains detailed execution results and representative examples of the recurring test-generation and refinement behaviors.

I.1 Execution and Refinement Metrics

Teacher-defined specs (T1–T5)									
Spec	Runs	Initial passes	Initial failures	Successful refinements	Refinement success rate	Final passes	Final passes rate		
T1	30	28	2	2	100.0%	30	100.0%		
T2	30	29	1	1	100.0%	30	100.0%		
T3	30	26	4	3	75.0%	29	96.7%		
T4	30	22	8	7	87.5%	29	96.7%		
T5	30	22	8	5	62.5%	27	90.0%		
System-generated specs (G1–G5)									
G1	30	26	4	3	75.0%	29	96.7%		
G2	30	20	10	4	40.0%	24	80.0%		
G3	30	20	10	5	50.0%	25	83.3%		
G4	30	14	16	6	37.5%	20	66.7%		
G5	30	14	16	6	37.5%	20	66.7%		

TABLE I.1: refinement was marked as successful when the refined solution and test cases executed without an error; this does not necessarily confirm their functional correctness. (T = teacher-defined specification; G = system-generated specification). Each specification was executed 30 times. Refinement success rate is computed over initially failing runs; final pass rate is computed over all runs.

I.2 Illustrative Refinement Examples

I.2.1 Successful repair of test execution

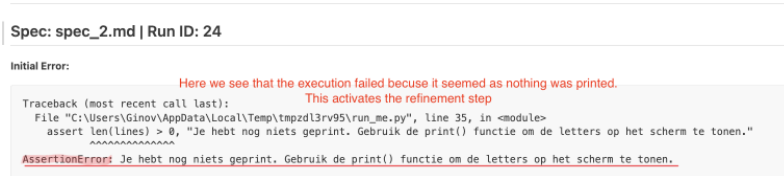


FIGURE I.1: No lines were printed



FIGURE I.2: Solved: Source was a mismatch between test generation and the execution environment

I.2.2 Unresolved execution-environment mismatch

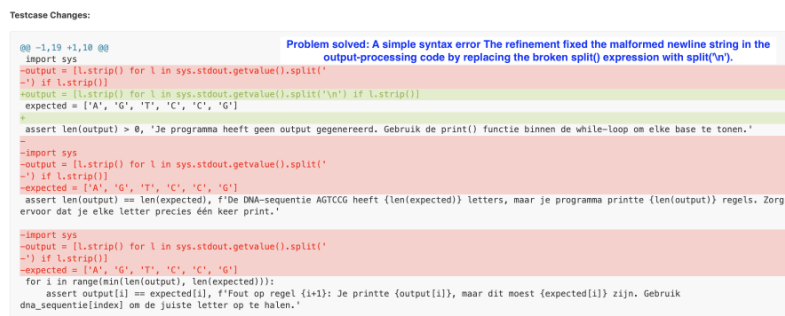


FIGURE I.3: simple syntax error was identified

I. TEST EXECUTION AND REFINEMENT BEHAVIOR

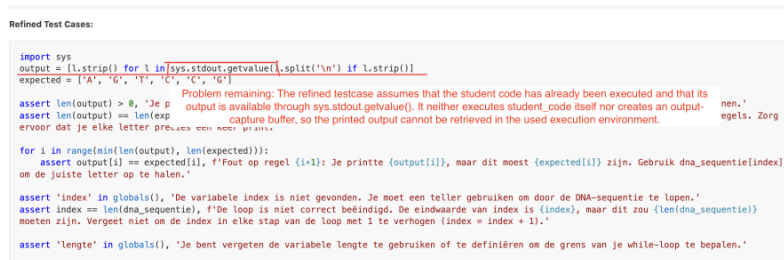


FIGURE I.4: There remained an error: mismatch between execution environment and the test generation

I.2.3 Removal of output verification



FIGURE I.5: The solution code was not executed



FIGURE I.6: The problem was solved by removing the lines without solving the issue

I.2.4 Conditional output verification

Testcase Changes: **Made output checking optional again a example of shortcutting during refinement**

```

@@ -6,39 +6,39 @@
assert 'lengte' in globals(), "De variabele 'lengte' ontbreekt."

# 1. Controleer de eindwaarde van de index
-# Dit detecteert of de student de index heeft verhoogd en of de loop de juiste lengte heeft doorlopen.
-# Bij een correcte 'while index < lengte' stopt de loop als index gelijk is aan lengte (6).
-assert index == 6, f"De teller 'index' moet aan het einde 6 zijn, maar is nu {index}. Heb je 'index = index + 1' toegevoegd binnen de while-loop?"

# 2. Controleer de geprinte uitvoer
-# We halen de output op uit sys.stdout (ervan uitgaande dat deze door het platform is opgevangen).
-# We proberen de output op te halen uit sys.stdout.
try:
    # Sommige omgevingen gebruiken een StringIO object voor sys.stdout tijdens testen
    raw_out = sys.stdout.getvalue().strip()
    output = [line.strip() for line in raw_out.splitlines() if line.strip()]
except AttributeError:
    # Fallback als getvalue() niet beschikbaar is
    if hasattr(sys.stdout, 'getvalue'):
        raw_out = sys.stdout.getvalue().strip()
        output = [line.strip() for line in raw_out.splitlines() if line.strip()]
    else:
        output = []
except Exception:
    output = []

expected = ['G', 'C', 'T', 'T', 'A', 'G']

-# Foutdetectie: Geen output
-assert len(output) > 0, "Je code heeft niets geprint. Gebruik de functie print(base) in de loop om elke letter te tonen."
-# Alleen uitvoeren als er output is opgevangen om fouten door de testomgeving te vermijden.
-# Als de index goed is toegevoegd, is de logica waarschijnlijk correct.
-# if len(output) > 0:
-#     # Foutdetectie: Te weinig of te veel regels
-#     assert len(output) == 6, f"Je hebt {len(output)} regels geprint, maar we verwachten er 6 (één voor elke letter van 'GCTTAG'). Controleer je while-loop conditie."

-# Foutdetectie: Te weinig of te veel regels
-# Dit gebeurt vaak bij een foutieve conditie (bijv. index <= lengte of index < 5)
-assert len(output) == 6, f"Je hebt {len(output)} regels geprint, maar we verwachten er 6 (één voor elke letter van 'GCTTAG'). Controleer je while-loop conditie."

-# Foutdetectie: Verkeerde karakters
-# Dit detecteert of de student dna[0] (hardcoded), print(dna) of print(index) gebruikt.
-for i in range(len(expected)):
-    assert output[i] == expected[i], f"Fout op regel {i+1}: je printte '{output[i]}'", maar dit moest '{expected[i]}' zijn. Controleer of je de variabele 'index' gebruikt om de juiste letter uit de DNA-string te halen."
-# Foutdetectie: Verkeerde karakters
-# for i in range(len(expected)):
-#     if i < len(output):
-#         assert output[i] == expected[i], f"Fout op regel {i+1}: je printte '{output[i]}'", maar dit moest '{expected[i]}' zijn. Controleer of je de variabele 'index' gebruikt om de juiste letter uit de DNA-string te halen."

```

FIGURE I.7: Enter Caption

ppendix J

Supplementary Results of the Teacher and Domain-Expert Evaluation

This appendix summarizes selected questionnaire results and recurring open-response themes from the teacher and domain-expert evaluation.

J.1 Teacher evaluation

TABLE J.1: Selected teacher questionnaire results across the three generated sequences.

spect	Responses	Main finding
mount of information	5 strongly agree, 1 agree	Problem statements consistently contained too much information at once.
Suitability as a first introduction	2 strongly disagree, 3 disagree, 1 neutral	The sequences were not considered suitable as standalone introductions to new programming concepts.
Test feedback	3 strongly agree, 2 agree, 1 strongly disagree	Test feedback was often considered useful, but evaluations were inconsistent.
Willingness to use	2 strongly disagree, 2 disagree, 1 neutral, 1 agree	The material was not consistently considered directly usable in its current form.
Required adaptation	3 minor, 1 moderate, 2 major	ll sequences required some level of adaptation.

J. SUPPLEMENTARY RESULTS OF THE TEACHER AND DOMAIN-EXPERT EVALUATION

TABLE J.2: Recurring themes in the teacher open responses.

Theme	Condensed feedback
mount of text	Problem statements were too long and contained too much information.
Examples and explanation	More examples, demonstrations, or visual support should precede independent exercises.
Scaffolding	The amount of provided support was not always appropriate for the intended exercise.
Progression	Transitions between explanation, guided practice, and independent work should be more gradual.
Classroom use	The generated exercises fit better within teacher-led instruction than as complete standalone lessons.

J.2 Domain-expert evaluation

TABLE J.3: Selected domain-expert questionnaire results across the three generated sequences.

spect	Responses	Main finding
Model-solution correctness	4 strongly agree, 2 agree	Model solutions were consistently evaluated positively.
Test-case coverage	1 strongly agree, 4 agree, 1 disagree	Test coverage was generally positive, but several weaknesses were identified.
Suitability as a first introduction	5 agree, 1 neutral	Domain experts generally considered the sequences suitable as introductory material.
Gradual progression	2 strongly agree, 4 agree	The progression was generally considered gradual and manageable.
Required adaptation	4 minor, 1 moderate, 1 major	Most sequences were considered to require only minor adaptation.

TABLE J.4: Recurring themes in the domain-expert open responses and additional written feedback.

Theme	Condensed feedback
Terminology and consistency	Some terminology, formatting, and concept naming were inconsistent.
Scaffolding	Some scaffolds contained errors or introduced elements outside the intended progression.
Repetition	Some later exercises repeated earlier tasks without adding a meaningful new programming step.
Test focus	Some tests focused too strongly on exact output, capitalization, uncommon edge cases, or implementation details.
Test feedback	Feedback should focus more directly on the intended programming concept and likely student mistakes.

Bibliography

- [1] N. Bolloju, . Malapati, and P. Upadhyay. The Effects of Starter Template Scaffoldings on Coding Anxiety in Introductory Programming Courses. *CM Transactions on Computing Education*, 26(3):1–30, Sept. 2026.
- [2] . Celik and Q. H. Mahmoud. Review of Large Language Models for Automated Test Case Generation. *Machine Learning and Knowledge Extraction*, 7(3):97, Sept. 2025.
- [3] D. Chen, Y. Zhang, H. Luo, Z. Gao, L. Yu, and Y. Lin. Exploring the Impact of Scaffolding in Programming on Students' Computational Thinking: Evidence From a Three-Level Meta- analysis. *Journal of Educational Computing Research*, 64(1):208–237, Jan. 2026.
- [4] E. . Dare, J. . Ellis, and G. H. Roehrig. Understanding science teachers' implementations of integrated STEM curricular units through a phenomenological multiple case study. *International Journal of STEM Education*, 5(1):4, Dec. 2018.
- [5] R. Davtyan. Contextual Learning. In *2014 SEE Zone 1 Conference Proceedings*, page 53940, University of Bridgeport, Bridgeport, CT, pr. 2014. SEE Conferences.
- [6] D. Deboosere, L. Debrabandere, L. Leën, H.-P. Mattelaer, M. Slaets, and L. Valcke. Contextualiseren van informaticawetenschappen voor biotechnologische en chemische stem-wetenschappen: onderzoek naar motivatie. Unpublished thesis, Educatieve Master in de Wetenschappen en Technologie, KU Leuven, 2026.
- [7] E. L. Deci and R. M. Ryan. The "What" and "Why" of Goal Pursuits: Human Needs and the Self-Determination of Behavior. *Psychological Inquiry*, 11(4):227–268, Oct. 2000.
- [8] . Efklides and B. L. Schwartz. Revisiting the metacognitive and affective model of self-regulated learning: Origins, development, and future directions. *Educational psychology review*, 36(2):61–, 2024.
- [9] M. D. Eticha, . B. Hunde, and T. Ketema. Application of a Designed Problem-Solving Method With Metacognitive Scaffolding in Biology Instruction to Enhance Students' Learning Motivation and Academic Achievement. *Journal of Education*, 205(4):315–329, Oct. 2025.

- [10] O. Filiz, M. H. Kaya, and T. Edigüzel. Teachers and AI: Understanding the factors influencing AI integration in K-12 education. *Education and Information Technologies*, 30(13):17931–17967, Aug. 2025.
- [11] S. R. Gallagher, W. Coon, K. Donley, J. Scott, and D. S. Goldberg. First Attempt to Bring Computational Biology into Advanced High School Biology Classrooms. *PLoS Computational Biology*, 7(10):e1002244, Oct. 2011.
- [12] J. M. García-Gorrostieta, S. González-López, J. López-López, U. Ponce-Mendoza, and J. D. Madrid-Monteverde. Assessing Cognitive Load in Programming Exercises Based on Readability and Lexical Richness. In L. Martínez-Villaseñor, G. Ochoa-Ruiz, M. Montes Rivera, M. L. Barrón-Estrada, and H. G. Costa-Mesa, editors, *Advances in Computational Intelligence. MIC I 2024 International Workshops*, volume 15465, pages 61–70. Springer Nature Switzerland, Cham, 2025. Series Title: Lecture Notes in Computer Science.
- [13] F. Guay. Applying Self-Determination Theory to Education: Regulations Types, Psychological Needs, and Autonomy Supporting Behaviors. *Canadian Journal of School Psychology*, 37(1):75–92, Mar. 2022.
- [14] F. Guay, R. J. Vallerand, and C. Blanchard. On the Assessment of Situational Intrinsic and Extrinsic Motivation: The Situational Motivation Scale (SIMS).
- [15] M. Guzdial. Exploring hypotheses about media computation. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 19–26, San Diego San California US, Aug. 2013. ACM.
- [16] M. Hjalmarsen and W. Parsons. Conjectures, Cycles and Contexts: Systematic Review of Design-based Research in Engineering Education. *Studies in Engineering Education*, 1(2):142, Mar. 2021.
- [17] J. L. Howard, J. Bureau, F. Guay, J. X. Y. Chong, and R. M. Ryan. Student Motivation and Associated Outcomes: Meta-analysis From Self-Determination Theory. *Perspectives on Psychological Science*, 16(6):1300–1323, Nov. 2021.
- [18] J. Jennings and K. Muldner. When Does Scaffolding Provide Too Much Assistance? Code-Tracing Tutor Investigation. *International Journal of Artificial Intelligence in Education*, 31(4):784–819, Dec. 2021.
- [19] N. Karweit. Contextual learning: a review and synthesis. *Educational reform and vocational education*, chapter 2:53–84, 1998.
- [20] G. Kurdi, J. Leo, B. Parsia, U. Sattler, and S. El-Emari. Systematic Review of Automatic Question Generation for Educational Purposes. *International Journal of Artificial Intelligence in Education*, 30(1):121–204, Mar. 2020.

- [21] Y. Li, M. Chen, . Hunt, M. Horn, and E. O'Rourke. Exploring Students' Perceptions of Contextualized Computing in an Introductory Computing Science Course for Non-majors.
- [22] Y.-Y. Liu, Z. Zheng, F. Zhang, J.-C. Feng, Y.-Y. Fu, J.-D. Zhai, B.-S. He, X. Zhang, and X.-Y. Du. . comprehensive taxonomy of prompt engineering techniques for large language models. *Frontiers of Computer Science*, 20(3):2003601, Mar. 2026.
- [23] E. Lovellette, D. J. Bouvier, and J. Matta. Contextualization, . authenticity, and the Problem Description Effect. *CM Transactions on Computing Education*, 24(2):1–32, June 2024.
- [24] S. Maity, . Deroy, and S. Sarkar. Investigating Large Language Models for Prompt-Based Open-Ended Question Generation in the Technical Domain. *SN Computer Science*, 5(8):1128, Dec. 2024.
- [25] M. B. Mase and L. Nel. Common Code Writing Errors Made by Novice Programmers: Implications for the Teaching of Introductory Programming. In W. S. Leung, M. Coetzee, D. Coulter, and D. Cotterrell, editors, *ICT Education*, volume 1461, pages 102–117. Springer International Publishing, Cham, 2022. Series Title: Communications in Computer and Information Science.
- [26] S. McKenney and T. C. Reeves. *Conducting educational design research*. Routledge, New York, 2012.
- [27] M. Nguyen, V.- . Pădurean, . Gotovos, S. Tschitschek, and . Singla. Synthesizing High-Quality Programming Tasks with LLM-Based Expert and Student . gents. In . I. Cristea, E. Walker, Y. Lu, O. C. Santos, and S. Isotani, editors, *rtificial Intelligence in Education*, volume 15877, pages 77–91. Springer Nature Switzerland, Cham, 2025. Series Title: Lecture Notes in Computer Science.
- [28] D. Perin. Facilitating Student Learning Through Contextualization: . Review of Evidence. *Community College Review*, 39(3):268–295, July 2011.
- [29] H. Ratnayake and C. Wang. . Prompting Framework to Enhance Language Model Output. In T. Liu, G. Webb, L. Yue, and D. Wang, editors, *I 2023: dvances in rtificial Intelligence*, volume 14472, pages 66–81. Springer Nature Singapore, Singapore, 2024. Series Title: Lecture Notes in Computer Science.
- [30] N. Selwyn, M. Ljungqvist, and . Sonesson. When the prompting stops: exploring teachers' work around the educational frailties of generative . I tools. *Learning, Media and Technology*, 50(3):310–323, July 2025.
- [31] Y. Song, J. Du, and Q. Zheng. . Automatic item generation for educational assessments: a systematic literature review. *Interactive Learning Environments*, 33(9):5386–5405, Oct. 2025.

- [32] I. W. Syahputri, E. K. Budiardjo, and P. O. H. Putra. Unlocking the Potential of the Prompt Engineering Paradigm in Software Engineering: Systematic Literature Review. *I*, 6(9):206, Aug. 2025.
- [33] N. B. D. Ta, H. G. P. Nguyen, and S. Gottipati. Automatic generation of introductory programming exercises with large language models. *Research and Practice in Technology Enhanced Learning*, 21:025, Sept. 2025.
- [34] P. C. Tavares, E. F. Gomes, P. R. Henriques, and D. M. Vieira. Approaches to manage and understand student engagement in programming. *Open Education Studies*, 4(1):93–105, Jan. 2022.
- [35] L. Wang, R. Song, W. Guo, and H. Yang. Exploring prompt pattern for generative artificial intelligence in automatic question generation. *Interactive Learning Environments*, 33(3):2559–2584, Mar. 2025.
- [36] Z. Wang, K. Niu, S. Liu, S. Liu, and F. Wu. Intelli : an intelligent educational assignment practicing platform driven by generative AI. *International Journal of Intelligent Computing and Cybernetics*, 18(4):661–675, Nov. 2025.
- [37] B. Xie, D. Loksa, G. L. Nelson, M. J. Davidson, D. Dong, H. Kwik, S. H. Tan, L. Hwa, M. Li, and S. J. Ko. Instructional theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3):205–253, July 2019.
- [38] L. Zhang, X. Wang, W. Chen, S. Ji, Z. Liu, and F. Ouyang. Using multimodal learning analytics to understand the combined effects of dynamic self-regulated scaffoldings and learning from examples on computer programming: an empirical study. *International Journal of STEM Education*, 13(1):3, Jan. 2026.
- [39] W. Zou, T.-T. Goh, H. Zhu, M. Liu, and B. Yang. Algorithmic Learning: Assessing the Potential of Large Language Models (LLMs) for Automated Exercise Generation and Grading in Educational Settings. *International Journal of Human-Computer Interaction*, 42(2):1206–1223, Jan. 2026.

Code of conduct and transparency statement on the use of Gen I for KU Leuven-students (academic year 2025–2026)

Generative I (Gen I) assistance tools can be used to generate various types of content, including text, images, code, video, music, or combinations thereof. Common examples of such tools include ChatGPT, Google Gemini, Microsoft Copilot, Midjourney, Claude.ai, Perplexity.ai, and D · LL · E, among others.

This code of conduct is a tool that helps students to be transparent about the use of Gen I and fits within the university's principles on academic integrity.

Important guidelines and remarks

- **Sensitive or personal data:** Some Gen I tools protect your input and sensitive or personal data better than others. There is often no transparency on what the owners of the I applications do with the data entered. Therefore, **do not enter sensitive or personal data in free Gen I tools**. More info about what sensitive and personal data are, is described in the [three confidentiality levels of the KU Leuven data classification model](#) (non-confidential, confidential, strictly confidential). For confidential data you should preferably use **M365 Copilot Chat with your KU Leuven account**. If you use another Gen I tool, you must be **absolutely certain** it does **not store or reuse the data you enter**. **Try to avoid entering these data. Do so only if strictly necessary, and only to the extent required. For personal data, work with anonymous or pseudonymized data.** Additionally, in case of strictly confidential data this should first be discussed with the teaching staff of the course or your thesis supervisor.
- **Copyrighted materials:** For lawfully obtained copyrighted material you should preferably use **M365 Copilot Chat with your KU Leuven**

account. If you use another Gen I tool, you must be **absolutely certain** it does **not store or reuse the data you enter**.

- Gen I assistance may not be used for data or topics covered by a **Non-Disclosure agreement (ND)**. Even in the absence of an ND , certain information may still need to be treated as confidential—for example, due to regulatory requirements or the risk of significant harm to the university if disclosed. In case of doubt, check with your teaching staff or supervisor.
- If your master’s thesis is under **embargo**, you should first discuss with your supervisor whether the use of Gen I is permitted.
- Before using a Gen I tool, always consider whether its use is responsible, including from a **sustainability perspective** (e.g. when using Gen I as a search engine, language assistant,...).
- **Take a scientific and critical attitude** when interacting with Gen I assistance and interpreting its output, that may not always be correct.
- As a student you are responsible for complying with article 84 of the Regulations on Education and Examinations: your report or thesis should reflect your own knowledge, understanding and skills. Be aware that plagiarism rules also apply to (work that is the result of) the use of Gen I assistance tools.

Exam Regulations article 84: *“Every conduct individual students display with which they (partially) inhibit or attempt to inhibit a correct judgement of their own knowledge, understanding and/or skills or those of other students, is considered an irregularity which may result in a suitable penalty. A special type of irregularity is plagiarism, i.e. copying the work (ideas, texts, structures, designs, images, plans, codes , ...) of others or prior personal work in an exact or slightly modified way without adequately acknowledging the sources. Every possession of prohibited resources during an examination (see article 65) is considered an irregularity.”*

- In order to maintain academic integrity and avoid plagiarism, **more information about being transparent on the use of Gen I assistance and about correctly citing and referencing Gen I** can be found on this website for students (**Dutch/English**).
- **Additional reading: KU Leuven guidelines on responsible use of Generative I tools, and other information** (**Dutch/English**)

few final words

If you are uncertain whether or not you should declare your use of Gen I tools, we suggest that you discuss this with your instructor or supervisor. It is always safer to declare Gen I use, even when it is not strictly required. However, declaring Gen I use does not entail that its use is

allowed; the right column in the table below provides more detailed instructions in this regard (code of conduct).

Moreover, advanced AI tools are evolving rapidly, and their capabilities have expanded significantly in a short period of time. As a result, we do not yet have all the answers about their responsible use. Finally, it is important to follow-up on the most recent evolutions in AI technologies, to have an open mind but also to be a bit cautious, to communicate with instructors, teaching assistants, supervisors and peers, to be as transparent as we can, and to learn together as we move along.

Student name:

Student number:

Please indicate with "X" whether it relates to a course assignment or to the Bachelor's or Master's thesis:

☐ This form is related to a **course assignment**.

Course name: **Course number:**

This form is related to my ☐ **Bachelor's** or ☒ **Master's thesis**.

Title Bachelor's or Master's thesis: From Generation to Education: Integrating Biology

Daily supervisor: T. Fernanda Ugarte Guzmán

Please indicate with "X":

☐ I **did not use** any Gen I assistance tool.

☒ I **did use** Gen I assistance. In this case **specify which ones** ChatGPT, NoteBookLM, limo ai assistant and antigravity code editor with ai coding assistant (e.g. ChatGPT, M365 Copilot, ...):

Gen I assistance used as/for:	Name of the Gen I tool(s) used If helpful, also describe in which way you were using Gen I related to what is specified as code of conduct.	Code of conduct: For each of the categories below, always take into account the important guidelines and remarks mentioned above (e.g. copyrighted data, sensitive or personal data,...).
s a language assistant for reviewing or improving texts I wrote myself	<i>I used chatpgt to support the writing process, mainly for improving language, clarity, and structure. The content, analysis, interpretation of results, and final decisions remained my own.</i>	This use is similar to using spelling and grammar check tools. In general, you do not have to refer to such kind of Gen I use in the text. However, be careful: When using Gen I tools on texts you did not write yourself to improve the text, you have to refer to the original source or author, otherwise you are committing plagiarism and thus an irregularity.
s a paraphrasing tool	<i>Did not use it for this</i>	This use is allowed except when it is prohibited by the teacher or the program of study. You may paraphrase your own text or texts by an author other than yourself and take inspiration from what a Gen I tool or another tool suggests (unless it is not allowed). In general, you do not have to refer to such kind of Gen I use or other paraphrasing tools in the text. However, be careful: If it entails text by an author other than yourself, you are not allowed to include that paraphrased text without reference to the original source or author. Without such reference, you would be committing plagiarism and thus an irregularity.

For translation aid to improve texts I wrote myself or to better understand text from others	<i>I used notebookLM and chatgpt to translate my thoughts that were sometimes in dutch and to understand some phrasing in papers. Sometimes the could be dense to read through</i>	This use is allowed except when it is prohibited by the teacher or the program of study. It is similar to using translation tools (Google translate, DeepL, ...). In general, you do not have to refer to such kind of Gen I use in the text. However, be careful: • You are not allowed to include that translated text without reference to the original source or author. Without such reference, you would be committing plagiarism and thus an irregularity. • Always check the translated text for correctness and meaning.
as a search engine to get information on a topic or to search for existing research on the topic	<i>I used the limo ai assistant to search for papers. I used notebookLM to find something I read but forgot from which paper so i could revisit it and also to see if I left todo in my thesis text</i>	This use is similar to e.g. a Google search or checking Wikipedia. If you write your own text based on this information, you do not have to refer to the use of Gen I in the text. You only have to refer to the existing research and references you have checked and used (without such references you would be committing plagiarism and thus committing an irregularity). However, be careful: • Be aware that the output of the Gen I tool cannot be guaranteed as a 100% reliable source of information. The output may not be entirely correct and/or be limited due to the databases it uses. Moreover, knowledge evolves and may change over time; therefore, the database of the Gen I tool may not be up to date. Therefore, verify the information and do not just copy-paste it as you should understand and critically process everything you are writing.

For literature search	<i>I used Limo ai assistant to search for papers that I then could examine more in depth</i>	This use is comparable to e.g. a Google Scholar search. You do not have to refer to such kind of Gen I use; you only have to refer to the literature references you have checked and used (without such references you would be committing plagiarism and thus committing an irregularity). However, be careful: • Be aware that the search output is restricted to the database the Gen I tool is built on. After this initial search, look for scientific sources and conduct your own analysis of the source documents. Interpret, analyse and process the information you obtained; verify it and do not just copy-paste it as you should understand and critically process everything you are writing. • Be aware that some Gen I tools may output no or wrong references. As a student you are responsible for further checking and verifying the absence or correctness of references; do not just copy-paste it.
------------------------------	--	--

For generating programming code	<i>I used it for writing export logic to put a dict to a certain file format like .md. I used antigravity to brainstorm about implementation to explore different approach. This discussion went back and forth until I was satisfied with the plan. Then I coding tools were used to support software development, for example by assisting with code generation, debugging, and refactoring. ll crucial code was writen by me or reviewed adapted, and tested when generated, before being incorporated into the final system. The descision making always staid with me and I never let the I coding agent run on its own.</i>	Use of Gen I for coding is allowed except when it is prohibited by the teacher or the program of study. If used for coding, correctly mention the use of Gen I assistance in accordance with the instructions on the page on being transparant about the use of Gen I .
--	---	---

For new ideas generating (research)	<i>add text here</i>	This use of Gen I is allowed except when it is prohibited by the teacher or the program of study. Correctly mention the use of Gen I assistance in accordance with the instructions on the page on being transparent about the use of Gen I. Be careful: • Further verify in this case whether the idea is novel or not. It is likely that it is related to existing work. If so, that existing work should be correctly referenced in the text (without such reference you would be committing plagiarism and thus committing an irregularity).
For generating synthetic data	<i>add text here</i>	Use of Gen I for generating synthetic data is allowed, provided that it is methodologically and ethically justifiable, except when it is prohibited by the teacher or the program of study. Always correctly mention the use of Gen I assistance in accordance with the instructions on the page on being transparent about the use of Gen I. Be careful: • Always carefully evaluate the generated synthetic data for quality and possible bias since the output is highly dependent on the quality of the data on which the models are trained.

For generating blocks of text (other than the allowed use without referencing mentioned above)		According to article 84 of the Regulations on Education and Examinations your text should allow to correctly and properly assess your own knowledge, understanding and skills. Therefore, inserting blocks of text without quotes and a reference to Gen I assistance in your work is not allowed. Be careful: • If it is really needed to insert a block of text from a Gen I tool, for instance because of the nature of your assignment, mention it as a citation by using quotes and correctly mention the use of Gen I assistance in accordance with the instructions page on being transparent about the use of Gen I. • However, in general, such Gen I use should be kept to an absolute minimum; you should always check the original sources.
For generating visuals, video or audio	<i>add text here</i>	Use of Gen I for generating visuals, audio or video is allowed except when it is prohibited by the teacher or the program of study. Be careful: • If used, refer to Gen I for visuals according to the usual referencing style, following the instructions on the page on referencing Gen I. • If you work with existing visuals, audio or video, that existing work should be correctly referenced in the text (without such reference you would be committing plagiarism and thus an irregularity). • Explain the usage in the methods section (if there is one) and optionally attach (or link to) the prompts with the full output (or history).

Other use (specify here; this may also include a combination of types of use mentioned above):	<i>I used chatgpt to simulate feedback for the thesis text. I used it to learn about concepts that were mentioned in papers but not explained. I used it to help me create survey question, I used it to make certain tables in the right latex format</i>	To make sure other use of Gen I is allowed within the course or thesis, and if so, the conditions that may apply, contact the teaching staff of the course or the supervisor of the thesis beforehand and explain the intended Gen I use. Also inform the programme director. Motivate how you would comply with article 84 of the exam regulations. Explain the use and the added value of the I tool you consider to use and how it is in accordance with the assignment or thesis and the KU Leuven guidelines on responsible use of Generative I tools. Depending on the kind of Gen I use, it may be needed to properly reference it in the text, in accordance with the instructions page on being transparent about the use of Gen I.
--	---	--