

École polytechnique de Louvain

Increasing the diversity of feedback types in a CS1 course

Author: **Charlotte VERSTRAETE**
Supervisors: **Kim MENS, Olivier GOLETTI**
Readers: **Kim MENS, Olivier GOLETTI, Tom BARBETTE**
Academic year 2022–2023
Master [120] in Civil Engineering

Contents

1	Abstract	3
2	Acknowledgments	5
3	Introduction	6
3.1	Context	6
3.2	Problem	7
3.3	Motivation	8
3.4	Objectives	9
3.5	Approach	9
3.6	Contributions	10
3.7	Roadmap	10
4	Background Material	12
4.1	Definition of feedback	12
4.2	The INGIInious Platform	12
4.3	The CS1 course	13
4.3.1	Contents	13
4.3.2	Structure	14
5	Analysis of the CS1 course	16
5.1	Analysis based on Keuning et al. [7]	16
5.2	Link with the classification of Ott et al. [6]	17
5.3	Classification of the INGIInious exercises	19
5.4	Problem Statement	22
6	Solution	24
6.1	Towards a solution	24
6.2	The Plans	26
6.2.1	Designing the diagrams	26
6.2.2	Accumulator	27

6.2.3	Accumulator With Condition	28
6.2.4	Filter	28
6.2.5	Maximum	30
6.2.6	Verification	32
6.2.7	Counter	32
6.2.8	Other implementations of pseudo-codes	34
6.2.9	Color Code	35
6.2.10	Similarities in Plans	37
6.3	Insertion in the course	37
6.3.1	Theory syllabus	37
6.3.2	INGInious exercises	39
7	Validation	44
7.1	Potential validation with Students	44
7.2	Validation with Tutors and Assistants	46
7.2.1	Method	46
7.2.2	Results	47
7.2.3	Analysis/discussion	48
7.2.4	Threats to validity	51
8	Final Discussion	52
8.1	Future Work	52
8.1.1	Before the exercise	52
8.1.2	During the exercise	53
8.1.3	After the exercise	54
8.1.4	Other Exercises	54
8.2	Conclusion	54

Chapter 1

Abstract

Learning the programming basics can be very challenging, as it demands a different way of thinking than students are used to. It requires the development of new skills and designing a course that allows to do this pedagogically can be difficult. At the Université Catholique de Louvain-La-Neuve, a course introducing programming for first year students was developed and has been taught in Python since five years. However, there is a constant desire to look for potential improvements and it is in this context that we carried out an analysis of the course.

The course is divided into eleven weeks. Each week consists of a lecture, two tutored sessions as well as individual work. The grading and feedback of the individual work partially goes through an auto-grader, INGINious. This auto-grading platform enables students to submit their code for the exercises and provides them immediate feedback. The feedback is not automatically generated by INGINious in itself, but depending on which tests pass, the error message the teaching staff has encoded varies.

We analyzed the diversity in types of feedback and concluded that there was a lot of feedback that focused on the task itself but not so much about the process of coding or about self-reflection. This represents a missed opportunity to help the students grow in autonomy and gain perspective. Since the course is organized in weeks, sometimes students also fail to see the link between the topics and exercises across the different weeks.

To remedy this problem, we identified patterns that recur in the solutions of the exercises. For this thesis we focused on the similarities between data structures, coming up in total with six different patterns or "plans". In order to make these plans clear to students the first step was to develop a conceptual flowchart for each pattern. These flowcharts are designed to explain the plan in general and to be

applicable to every data structure they will encounter in this introductory programming course. To provide a more practical aspect to these patterns, pseudo-codes have been developed, illustrating them. They are data structure-specific but stay easy to adapt to different types of problems.

In the course, this was integrated directly in the theory syllabus, exposing these patterns as students go along the course. Several pseudo-codes for each pattern are given to illustrate how to implement them for the different data structures. Some other ways of implementing the same pattern for the same data structure are also demonstrated, for example by using a while loop instead of a for loop. The integration of these plans in the exercises was done in three different ways, ranging from a lot of help to just an indication. The first is scaffolding: the structure of the plan of the exercise is given, leaving some blanks for the student to fill. Another way of helping the students is by giving them hints on which pattern this specific exercise uses. Finally, the last way this can be integrated is to make explicit links between exercises - "this exercise can be solved similarly to the exercise(s) X".

As the timing did not allow these plans to be evaluated with students, a validation with tutors was put in place. The patterns were quite well received and the tutors generally felt that the objectives of the plans had been achieved. The plans were found to be especially effective in helping students to start a new exercise. However, the link between the diagrams, on the one hand, and between the exercises sharing the same plan, on the other, is still not very clear and needs to be improved. Other ways of increasing the diversity of feedback were also identified for future work.

Chapter 2

Acknowledgments

During the development of this thesis, I was fortunate to receive advice, support and encouragement from many people who played an important role in its development. This chapter is dedicated to expressing my gratitude to those who contributed to the realization of this thesis.

First of all, I would like to express my sincere gratitude to Kim Mens and Olivier Goletti for their constant availability and patience during our frequent meetings. Their precious advice and guidance have played a decisive role in my progress throughout the year.

I would also particularly like to thank the tutors and assistants who generously took part in the survey and provided valuable information for evaluating my work. Their active involvement greatly contributed to the assessment of my research.

Finally, I'd like to express my sincere thanks to my family and friends, who have often been good listeners and given me good advice. They motivated me throughout this thesis and helped me bring this project to fruition.

Chapter 3

Introduction

3.1 Context

Learning how to code can be challenging at first as it requires a specific mindset and way of thinking that typically comes with experience. A lot of the mechanisms behind the coding of senior programmers are acquired by practice and a lot of times implicitly. The steps and way of thinking for resolving specific issues come step by step with time. For example, calculating the sum of an array seems very trivial, but beginners first need to know what arrays are, how to iterate over them as well as how to sum every element of this array. Once this is acquired, programmers do not always think of these as individual steps anymore. As such, it can be difficult for them to put themselves in the shoes of fresh students when explaining those concepts.

The three introductory programming courses at UCLouvain, *LINFO1401 - Informatique 1*, *LINFO1101 - Introduction à la programmation* and *LSINC1101 - Introduction à la programmation*, were given in Python for the first time in the academic year 2018-2019. However, prior to this, an equivalent course had been given in Java for several years. It allows students to discover the basics of programming, covering concepts such as loops, functions or file processing, as well as data structures like arrays, dictionaries or linked lists. Students also get to understand the basic concepts of object-oriented programming, learning about classes, inheritance and polymorphism. This course will now be referred to as a CS1 course in the thesis, as it is commonly called in the literature.

The course is structured around a weekly topic and specific exercises to do. Each week there is a specific problem and a recurrent study pattern the student is supposed to follow. It includes a lecture, tutored sessions but also a lot of exercises

the students are expected to solve individually or in small groups. These exercises are almost always described and evaluated through the INGINious platform.

3.2 Problem

The CS1 course is vital for building programming fundamentals that students need throughout their curriculum. However, improvements are constantly needed. The challenge is even greater since many first-semester university students are unfamiliar with the learning environment, making it easy for them to feel lost. Additionally, the teaching staff may struggle to understand students' difficulties as the course covers basic concepts.

It is in this context of improvement that we wanted to evaluate the quality of the course, and more particularly the quality in diversity of feedback in INGINious exercises. In *A Systematic Literature Review of Automated Feedback Generation for Programming* [7], Keuning et al. describe a classification of different types of feedback in order to compare different automated feedback generation tools. We applied the same classification on the exercises of the CS1 course, the result of which is presented on Figure 3.1. This allowed us to bring to light some imbalances in the type of feedback. Indeed, on Figure 3.1 we notice that there is a lot of feedback regarding Knowledge about Task Constraints (**KTC/TR**: feedback specifying the goal and/or task requirements of an exercise) and Knowledge about Mistakes (**KM**: feedback about the student's mistakes) but barely any feedback regarding Knowledge about How to proceed (**KH**: hints on how to proceed to solve the problem) or Knowledge about Concepts (**KC**: explanations about the concepts and knowledge behind the exercise). The feedback type of Knowledge about Meta-Cognition (**KMC**: feedback encouraging the student's awareness of its capacities and task progression) is not even represented, as there was no evidence of such feedback in the exercises. Following the classification described in Ott et al. [6], this could also be expressed as a lot of feedback on Task Level but a lack of feedback at a Process level or Self-regulation level. Feedback at a Process level focuses on the process behind solving a solution rather than the solution in itself, while feedback at a Self-regulation level focuses on encouraging the students to self-monitor.

Ott et al. [6] praise the effectiveness of feedback on a self-regulatory level, and mention some ways to implement this in a typical CS1 course. While they did not fit our categorisation, the majority of these ideas are already implemented in the CS1 course. Therefore, we prioritized emphasizing Process level feedback, as it was

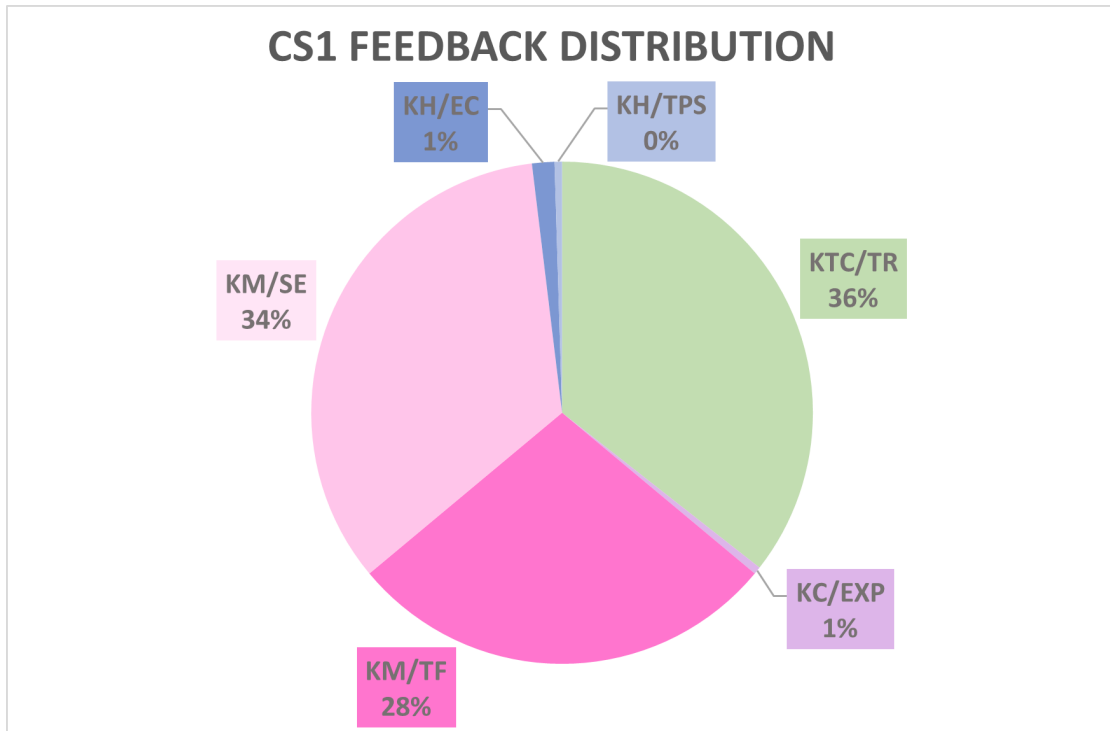


Figure 3.1: Distribution of the feedback types in the CS1 course

not yet fully implemented and thus had a greater potential impact than already implemented feedback. Ott et al. [6] also highlight the effectiveness of feedback at the Process level, stating it is even more effective than feedback on task level. According to Keuning et al. [7], this aligns with providing feedback About How to Proceed.

3.3 Motivation

This bias in the type of feedback as well as the general structure of the course have an impact on how students approach the exercises. Often, students are very focused on the topic and the exercises of the week, failing to see connections between the topics covered in different weeks. A practical example could be the similarities in use of data structures such as arrays and dictionaries, or the fact that loops can be used to repeat expressions as well as to iterate over data structures.

This prevents students from comprehending the big picture and slows down their progress in coding and general understanding. When considering only one topic, they fail to see how a solution of a previous week could be adapted to the new

problem. For example, in week 4 an exercise "Ajoute" can be solved similarly to "Prime Number", an exercise from week 2. By approaching the second problem individually, they can feel lost and may not know how to proceed. This can discourage them to try as they don't even have a clear basis to start with.

3.4 Objectives

Through learning about the different types of feedback and the quality of feedback, as well as through personal reflections we identified a series of ways to implement different kinds of feedback, which will be detailed more later on. After some thinking, we decided to put the emphasis on giving students indications on how to proceed.

One way to remedy the problem would be to identify similarities between the solutions of exercises. Showing students how to solve a particular sub-problem would give them something to start with and a clear way of how to proceed. This also helps them apply similar solutions to slightly different problems.

The goal of this thesis is thus twofold. The first is to help students have some basis to start with when they feel lost in an exercise. We want to create a reference they can look for, and promote an active thinking of the process. The second objective is to encourage learning transfers between exercises. By identifying recurring patterns between exercises, students will be more likely to solve new problems based on previous ones.

3.5 Approach

In order to provide more feedback regarding Knowledge about How to proceed, we wanted to establish links and identify the similarities between different solutions. The aim is to provide a general outline for students to follow. We first tried to find coding patterns between the solution of exercises. Inspired by the plans developed by W. Lewis Johnson for PROUST [1], we identified a set of plans that are used in different exercises across the course. Since the plans developed by W. Lewis Johnson did not have the same objective as we had in this thesis, we had to redefine them, making some of them more specific and others more general. Indeed, the goal of W. Lewis Johnson was to develop a tool that would understand the intentions of students by looking at their (sometimes buggy) code. Our objective however, was to uncover the correct plans to make students aware of the similarities between the

exercises, helping them project the solution of an exercise onto another.

3.6 Contributions

After finding a balance between designing patterns that are too general or too specific, we ended up with six different plans: *Accumulator*, *Accumulator with Condition*, *Verification*, *Filter*, *Counter* and *Maximum*.

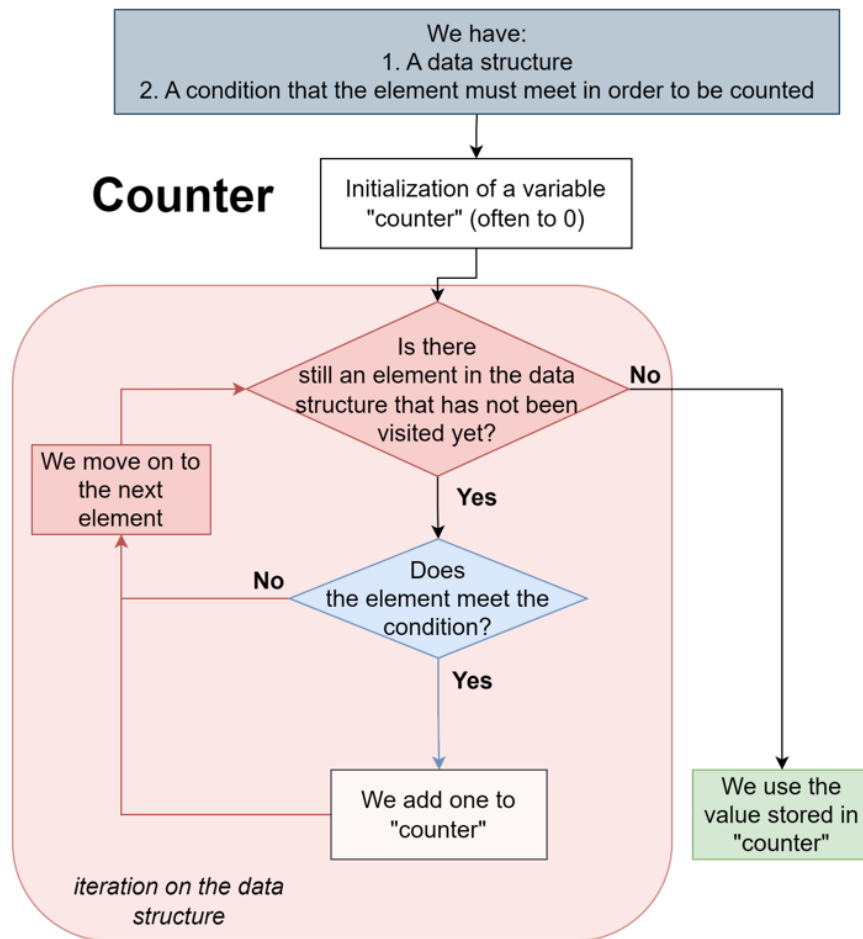
For each of these plans, we created a flowchart explaining the thinking behind each of these plans. By keeping a clear color code, the student can easily understand and use a new plan as well as spot the similarities between them. For each diagram, a pseudo-code is added to show the student how to practically implement the plan. There are multiple pseudo-codes for each flowchart as they depend on the data structure and implementation choices. An example of a flowchart and an associated pseudo-code is illustrated on Figure 3.2.

These plans were then inserted in the course in multiple ways. First they were presented in the theory syllabus, by the means of some examples. These plans were also and mainly used in the INGIInious exercises, as hints or with scaffolding. Hints were implemented in two ways, as hints about the links between exercises or straight up hints about which plan this exercise is based on.

3.7 Roadmap

The remainder of this document is structured as follows: the next chapter will provide the necessary background material, where our definition of feedback will be defined. There will also be an explanation and a contextualization of the INGIInious Platform as well as the CS1 course. Next we will detail our analysis of the course, following two classifications. In this chapter a mapping between both categorizations will be proposed as well as the method used for applying this to our course. The result and early conclusions of this analysis will be displayed too. This will lead to the problem statement, where we will conclude that there is a need for more feedback at a Process level. The development of the plans necessary to implement this, as well as the design of the resulting diagrams will be discussed in the first sections of the Solution chapter. Their insertion in the course, both in the theory syllabus as in the INGIInious exercises will be discussed at the end of that chapter. Its effectiveness will then be judged according to a validation with tutors. Finally, in a section "Future work", we will discuss other possible solutions to diversify the types of feedback in the INGIInious exercises and we will conclude

with a section in which we reflect on the work done and summarize this thesis.



(a) Flowchart of the Counter plan

```

a_list = (some predefined list)
counter = 0
for element in a_list:
    if condition(element):
        counter = counter + 1
return counter

```

(b) A pseudo-code of the Counter plan for lists

Figure 3.2: Example of the Counter plan

Chapter 4

Background Material

4.1 Definition of feedback

In this thesis, the notion of **good** feedback will be used a lot. Sadler stated in his article *Formative assessment and the design of instructional systems* [2] that in order to speak about good feedback these three conditions have to be fulfilled, it needs to: "*(a) possess a concept of the standard (or goal, or reference level) being aimed for, (b) compare the actual (or current) level of performance with the standard, and (c) engage in appropriate action which leads to some closure of the gap*". This definition has also been used by Ott et al. [6] in their article, and is also the definition we will use throughout this thesis.

4.2 The INGINious Platform

In this thesis, we will mention the INGINious autograder a lot as well as the exercises designed on that platform. In order to understand how the developed diagrams will fit in the exercises, it is necessary to understand the context and purpose of INGINious.

INGInious is a web-based platform where students can submit their code for exercises that teachers have created. The platform then runs the students' code and gives a grade, based on tests the teaching staff has designed. It gives feedback and allows students to correct their code until it passes all the tests. Students can then see in real time if their code compiles and passes (a part of) the tests. INGINious is also used to conduct exams.

As described on the site of UCLouvain [9], this tool was designed in a context of

providing a more pedagogical approach to coding. The teaching method followed an approach where students would individually solve problems and exercises. This demanded a lot of work from the tutors and assistants to correct and give feedback to every student individually. A tool was thus required to overcome the scaling problem regarding the number of students and the time it would take tutors to correct each submission. INGINious is now used in many courses at UCLouvain, in 2019 it was used by almost 3000 students and it evaluated 800.000 submissions.

4.3 The CS1 course

Adapted to the Python language five years ago, the CS1 course is an first year introductory programming course, given by Kim Mens, Siegfried Nijssen and Charles Pecheur. The theory syllabus is based on the open-source book *"How To Think Like a Computer Scientist: Learning with Python 3"* [3].

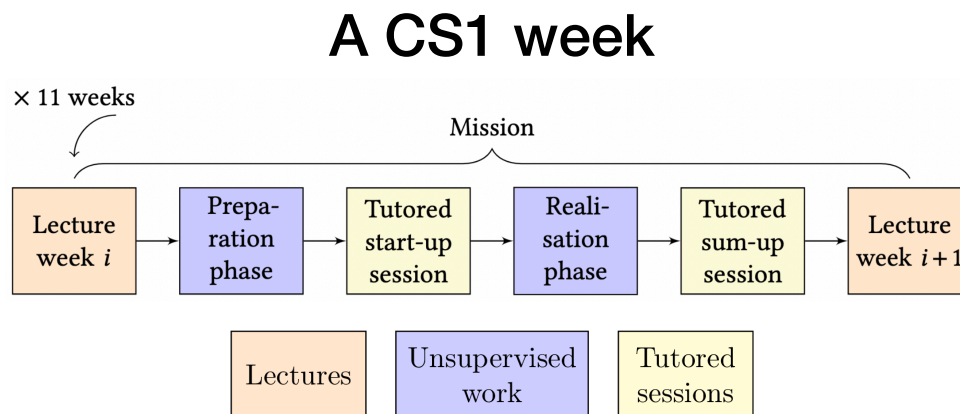
4.3.1 Contents

The weekly topics are distributed over eleven iterations, covering the basics and some data structures as well as classes and inheritance. The complete program of the course is as follows:

- Mission 1 - Getting Started
- Mission 2 - Programming basics
- Mission 3 - Functions and subproblems
- Mission 4 - Strings and lists
- Mission 5 - Tuples, Tests and Search Algorithms
- Mission 6 - Files and exceptions
- Mission 7 - Dictionaries
- Mission 8 - Classes and objects
- Mission 9 - Inheritance
- Mission 10 - Polymorphism
- Mission 11 - Linked lists

4.3.2 Structure

The course follows a weekly structure with a designated topic and corresponding exercises. Each week, students are presented with a specific problem and are expected to adhere to a recurring study pattern. It starts with a theoretical course, where the new topic of the week, linked to the current problem, is introduced. This is followed by autonomous learning where the student is expected to read a part of the theory syllabus as well as solve some exercises. In small groups, students then discuss this with a tutor who can answer all of their questions. During the realisation phase, they get some time to work in pairs to finally solve the problem of the week. Another session with the tutor is then scheduled, allowing him or her to give some feedback on the week's mission as well as answer the questions of the final task they solve during that session. The iteration eventually ends at the next theory class with a restructuring part, before starting the new topic. This specific structure is depicted on Figure 4.1.



1

Figure 4.1: Weekly structure of the CS1 Course; image taken from "*An Analysis of Tutors' Adoption of Explicit Instructional Strategies in an Introductory Programming Course.*" by Goletti, O., Mens, K. & Hermans, F. [8].

During the week, there are different sort of exercises the student can solve:

- Exercises in the exercise syllabus: These are not automatically graded and are corrected/solved with a tutor.
- Exercises on the INGINious platform : Exercises the student can work on autonomously since their solution is automatically graded, often with a

constructive feedback.

- Mandatory: The student is expected to solve all these exercises of a week during that week.
- Optional Exercises: Exercises for further study. They are indicated with the prefix Q^* .

Chapter 5

Analysis of the CS1 course

The first step of this thesis was to analyze the state of the course in terms of types of feedback. We based our analysis on two articles we already talked about previously: Keuning et al.'s [7] and Ott et al.'s article [6].

5.1 Analysis based on Keuning et al. [7]

Like the title suggests, in their article, "*A Systematic Literature Review of Automated Feedback Generation for Programming Exercises*", Keuning et al. [7] compare different ways to generate automated feedback for programming exercises. During the process, they try to answer the question "What is the nature of the feedback that is generated?", leading to a classification system of different types of feedback.

In their classification, based on the one described by Narciss [5], they distinguish two types of generated feedback: Simple feedback components and Elaborated feedback components.

Simple feedback components are divided into three categories: Knowledge of Result/response (**KR**), Knowledge of Performance for a set of tasks (**KP**), and Knowledge of the Correct Results (**KCR**). Knowledge of Result/Response provides information on the accuracy or validity of a solution for one task whereas Knowledge of Performance gives feedback about the performance after executing multiple tasks. Knowledge of Correct Results gives an explanation or indication of a valid solution.

Elaborate feedback components are divided into five categories which are almost all divided into sub-types. The first category is Knowledge about Task Constraints (**KTC**), which includes all feedback types that give more information on what the task is about. There are two categories: one that gives more explanation on what

is required, "Hints on Task Requirements" (**TR**) and one that focuses on how to approach the task, "hints on Task-Processing Rules" (**TPR**). Another category, "Knowledge about Concepts" (**KC**), gives feedback about the concepts necessary to solve the problem, by giving explanations on the subject matter (**EXP**), or by examples illustrating the concepts (**EXA**). The category Knowledge about Mistakes, is divided into five sub-categories: Test Failures (**TF**) indicating which tests fail, Solution Errors (**SE**) identifying errors in the behaviour of the solution, Compiler Errors (**CE**), Style Issues (**SI**) and Performance Issues (**PI**). Overall, this feedback type focuses on what the students does not do optimally or wrong. Another category is "Knowledge about How to proceed", (**KH**), with three sub-types. The first one, bug-related hints for Error Correction (**EC**), are hints on how to correct the solution. As opposed to hints on Task-Processing Rules, it takes into account the student's progress. The sub-type Task-Processing Steps (**TPS**) gives hints on the next steps to arrive to a solution. Once this solution is found, the feedback of sub-type Improvements (**IM**) gives indication on how to ameliorate this solution. Finally, the last category is Knowledge about Meta-Cognition (**KMC**), which involves a student's ability to recognize which strategy to use when solving a problem, their awareness of task progress, and their understanding of task execution quality.

5.2 Link with the classification of Ott et al. [6]

A feedback classification is also described in Ott et al.'s article [6], in which they discuss the categorization of Hattie et al. [4]. In this categorization, the feedback types are divided into four categories:

- Task level: how well the task has been understood
- Process level: targets the process to solve a task
- Self-regulatory level: becoming independent to solve tasks
- Self level: targets the student's self esteem

The latter however, according to Ott et al. [6], does not have a real impact on the performance of the student and is thus discarded for the rest of the discussion.

In his article, Ott et al. also gives the three main questions to ask at each level:

- Where am I going? (*"feed up"*)
- How am I doing? (*"feed back"*)

- Where to next? (*"feed forward"*)

the answer to which describe respectively the goals, actual progress/performance and the activities for improvement.

In order to represent the classification of Keuning et al. [7], we mapped those types of feedback into the classification of Ott et al. [6]. The result of this mapping is displayed in the following table. Highlighted in Green are the feedback types that are the most represented while the types highlighted in red are the ones we will end up trying to increase the presence of.

	Goals "feed up"	Progress/Performance "feed back"	Improvement "feed forward"
TASK	KCR KTC/TR KC/EXP KC/EXA	KP KR KM/TF KM/CE KM/SE	KM/SI KM/PI
PROCESS	KTC/TPR	KH/EC KH/TPS	KH/IM
SELF-REGULATION		KMC	

Table 5.1: Table representing the mapping of the feedback types used in Keuning et al. [7] to the feedback types used in Ott et al. [6]

On the task level, among the feedback types of Keuning et al. [7] that help identify the goals there is Knowledge About Correct Results and Hints on Task Requirements. Indeed, if the student cannot identify the goals of the task, that is if he does not know what the program is supposed to do (KCR) or does not know what exactly is expected from him (KTC/TR), these types of feedback can be very useful. In the same sense, if there are gaps in the student's knowledge (KC), it has to be addressed before the task can even be started.

The types of feedback relative to the Actual Progress of the student include the Knowledge of Performance for a set of tasks, Knowledge of Result/response and Knowledge about Mistakes. The types of Knowledge of Performance for a set of tasks and Knowledge of Result/response are included in this category as they are direct responses on the student's progression regarding the task. The Knowledge about Mistakes are also an indication on what the student has to change in his

code to arrive to a solution, or to the ideal solution in the case of Performance Issues and Style Issues, which are considered an improvement at this point of the students' programming level.

Regarding the Process Level, hints on the Task-Processing Rules help the student get started on the task. This type of feedback has been included on the process level as the emphasis lays on the approach of the task. Still on the Process Level, but as "feed back" we find two subcategories of Knowledge about How to proceed: bug-related hints for Error Correction and Task-Processing Steps. They consider the student's progress but do not focus on the task itself, rather on the progress to solve the task. Finally, as activities for improvement we find the feedback type Improvements, which gives hints on how to improve a solution.

The last feedback type of Keuning et al. [7], Knowledge about Meta-Cognition, can be found on the Self-regulation level. Indeed, this type of feedback helps students take a step back, reflect on their task advancement and helps them monitor their own progress.

Some feedback types could fall into multiple categories. The Style Issues could for example be put in the activities for improvement at the task level, depending on if one considers the style to be an improvement or a necessity. In our case, we decided to classify this as an Improvement since this is a basic course. The same applies to the Performance Issues, as performance is not really discussed in this basic programming course.

5.3 Classification of the INGIInious exercises

To evaluate the overall heterogeneity of the automatic feedback given to students in our CS1 course, we classified the different types of feedback found during our analysis of INGIInious exercises into the categories discussed by Keuning et al. [7]. For this classification, we only included feedback from the mandatory INGIInious exercises (cfr Background Material), omitting the Multiple Choice Questions. Indeed, often these MCQ's did not fit a particular feedback type as Keuning et al. [7] described this classification specifically for coding. The results of this classification can be observed on Figure 5.1. It lists each type of feedback and its abbreviation with, in the "Absolute" column, the number of times feedback of this category has been encoded in INGIInious exercises. The "Relative" column shows the relative frequency compared to the total number of feedback (227), in percent. For clarity purpose, a color was given to each category. Sub-types were given a color in the

Abbreviation	Feedback Type	Absolute	Relative
KP	Knowledge of performance for a set of tasks	N.A.	N.A.
KR	Knowledge of result/response	N.A.	N.A.
KCR	Knowledge of the correct results	N.A.	N.A.
KTC/TR	Hints on task requirements	74	32.599119
KTC/TRP	Hints on task-processing rules	0	0
KC/EXP	Explanations on subject matter	1	0.4405286
KC/EXA	Examples illustrating concepts	0	0
KM/TF	Test failures	58	25.550661
KM/CE	Compiler errors	N.A.	N.A.
KM/SE	Solution errors	71	31.277533
KM/SI	Style issues	0	0
KM/PI	Performance issues	0	0
KH/EC	Bug-related hints for error correction	3	1.3215859
KH/TPS	Task-processing steps	1	0.4405286
KH/IM	Improvements	0	0
KMC	Knowledge About Meta-cognition	0	0

Figure 5.1: Table of the number of feedback per category

same color family.

A Hint on Task Requirement could be *"Nice try! But the operator '%' has been disabled. Please write an actual algorithm calculating the remainder."* (Q* Reste d'une division entière, Week 2), or a reoccurring one: *"You did not provide the requested method."*, (like in Maximum, Week 3). The only Explanation on subject matter was found in an exercise of the last week and was *"By using insert on an empty LinkedList, the item will become head of the LinkedList."*, (LinkedList - insert, Week 11). An example of a Test Failure feedback (KM/TF) is *"The median of {} is {} and you returned {}."*, (Median, Week 1) and an example of a Solution Error (KM/SE) is *"You don't properly process words that don't have common lines."* (Realisation Phase, Week 7). One of the feedback of bug-related hints for Error Correction (KH/EC) is *"Are you deleting the punctuation surrounded by spaces? Like this example: 'And now: a test!'"*, (Realisation Phase, Week 7). Finally, the only feedback of the category of task-processing steps is *"You are using a global variable which is not recommended; try to adapt your exercise to use a class variable instead of a global variable."* (Ticket - Variable de classe, Week 9).

Some categories were not included in the classification of the INGINious exercises because they did not apply. For clarification purpose they are showed in the table

as N.A.. In the case of the Simple feedback components, *Knowledge of performance for a set of tasks* and *Knowledge of result/response* are inherently integrated into INGINious as it always displays how many tasks pass the tests. *Knowledge of the correct results* never applies, as (indications of) solutions are only given during the session with the tutor.

For Elaborate feedback components, the compiler errors are also always embedded in the INGINious platform. Indeed, when the program does not even compile, INGINious automatically displays the compiler errors.

When counting we decided to count feedback messages that were too similar as one for the same exercise. The following two examples of the category *Knowledge about Mistakes - Test Failures (KM/TF)* were for example counted as one:

- The result of the program for data 1 gives X but you returned Y
- The result of the program for data 2 gives X but you returned Y

It is important to note that some feedback messages did not belong clearly to a particular category, as there was some room for interpretation. For example, the difference between the subcategories Solution Error and Test Failure of Knowledge about Mistakes was not always explicit. Indeed, the failure of a test indicates as well that the program does not behave like it is supposed to. We decided to put all feedback messages that were in the form "*You return X instead of Y*" in the Test Failure category and the rest in the Solution Error category.

The most frequent feedback messages were "*The result of {} is {} and you returned {}.*" a feedback of type knowledge of mistakes (KM/TF) and "*You did not name the method as expected.*", a feedback of type knowledge of task constraint (KTC/TR).

Thanks to this analysis we can observe that some categories of feedback are well represented while others are completely neglected. The three categories which combined form 97% of the feedback messages are hints on Task Requirements, Knowledge about Test Failures and Knowledge about Solution Errors. Unfortunately there is almost no feedback regarding Knowledge about Concepts, Knowledge on How to proceed or Knowledge about Meta-Cognition.

The categories that are already very present in the current course are highlighted in green in the table linking the classification of Keuning et al. [7] to the one of Ott et al. [6] (Table 5.1). We can observe that there is almost exclusively feedback on the Task level.

5.4 Problem Statement

Since the CS1 course lays the basics of programming that are necessary throughout the student's curriculum, there is a constant need to improve this course. New studies and articles emerge, raising relevant remarks on how to improve this type of course, and can be incorporated into the course. The possibilities of improvement are especially high since the course has only been taught in Python for a couple of years. There has not yet been time to fully explore every option of improvement.

A challenge the development of this course also faces is the fact that the majority of the students are in their first semester at the university and are not yet acquainted with how a university works. They can thus easily be lost in this new way of teaching. Additionally, as this course deals with the basics, it may also be hard for the teaching staff to understand the students' difficulties.

For this thesis, we put the focus on the INGIInious exercises, and more particularly on the feedback given. To evaluate the overall quality of the feedback, we performed the analysis previously detailed. During this analysis, we noticed that there was a lot of feedback of some types like Knowledge about Task Constraints or Knowledge about Mistakes. Although these types of feedback are necessary, a diversity of feedback types is desirable. As such, the lack of feedback of types Knowledge about How to proceed or Knowledge about Concepts for example is a problem, or at least a missed opportunity. Indeed, different types of feedback target different areas and have a different impact. If we look at the categorisation of Ott et al. [6], this feedback is equivalent to feedback on a Task level, which is good. But it is a missed opportunity, as it would be beneficial to encourage students to think about their process of solving an exercise or to encourage their autonomy.

In their article, Ott et al. remind us that according to J. Hattie and H. Timperley [4], feedback on a Process level *"is more effective than feedback on task level because it is targeting the development of task strategies rather than the task outcome"* [6]. They make a distinction between programming process and course-related process. The programming process refers to the specific steps involved in writing code and developing a program, while course-related processes encompass broader aspects of the CS1 course, including attendance, meeting requirements, and overall engagement with the course material. In this thesis we will solely talk about the first one, as we focus on the INGIInious exercises.

The feedback provided at the self-regulation level aims to enhance students' ability to monitor their own progress and effectively manage their learning process. It focuses on developing students' skills in self-assessment, self-reflection, and self-

directed learning. According to Ott et al. [6], feedback on a self-regulatory level can *"help them become confident and efficient software developers who know the tools of the trade and how to approach the problems involved"*. In the same article, they mention a couple of ways to implement this in a CS1 course. This includes Multistage Assignments or Self-Monitoring Course Performance, which are already implemented in the course. It did not show in our analysis because it was based on solely on the direct feedback in the exercises.

Since implementing self-regulation directly in the exercises did not fit very well and was also already partially implemented, we decided to put the emphasis on feedback on a Process level. Additionally, adding feedback on that level would make the most impact since it was not yet implemented at all. In the categorisation of Keuning et al. this would largely correspond to Knowledge about How to proceed.

A lack in feedback at a Process level can impact how students approach these exercises. By only focusing on the task at hand, they fail to see the connection between the exercises and between the topics of the different weeks. The objective of this thesis will be to demonstrate that the way of resolving different exercises can be very similar. The focus will lie in showing the similarities in different data structures, and in particular for iterating over them.

Chapter 6

Solution

6.1 Towards a solution

In order to provide more feedback of type Knowledge about How to proceed, we tried to identify the similarities between the INGIInious exercises of the course. We ended up identifying recurring "plans" or "patterns", used in several exercises. To find and select them we were inspired by some of the plans developed by W. Lewis Johnson for PROUST [1].

Johnson developed these plans in a very different context. He developed a program PROUST that would try to understand the students' intentions behind their coding and uncover their possible misconceptions or errors. PROUST was given a series of possible plans (both correct and erroneous), which it could adapt to the students' codes. This was developed for the programming language Pascal.

Because of this very different context, we had to adapt the plans to fit our needs. Some of Johnson's plans were incorrect precisely because they were designed to also recognize buggy codes. Other patterns just did not apply for our case. This was for example the case for the "Code Block" plan, which is just a begin-end pair, something specific to Pascal. The remaining plans also needed to be adapted, as they were sometimes too specific or not specific enough. For example, his "Average Plan" was too specific, and his "Condition if test" was too general for our purpose. As we will see later, "Average Plan" will end up as a possible example for our "Accumulator" plan. The "Condition if test" is too specific as *if* statements are omnipresent. Although we did not really split this into multiple categories, *if* statements are part of almost all final plans. The greatest challenge was to find a good balance between identifying plans that were neither too specific nor too general. To assess this, our criterion was the number of applications in the exercises:

if a potential plan was identifiable in almost all exercises, it had to be split into more specific plans. On the other hand, if it only occurred in one exercise, it meant the plan was too specific. An exception however was made for the Maximum Plan, which appeared only a couple of times in the exercises but could not be more abstract at the risk of being too general.

In total, six plans were identified: *"Accumulator"*, *"Accumulator With Condition"*, *"Verification"*, *"Maximum"*, *"Filter"* and *"Counter"*. Only Maximum and Accumulator really coincide respectively with Johnsons' Maximum Plan and Running Accumulation, the rest is adapted or totally added.

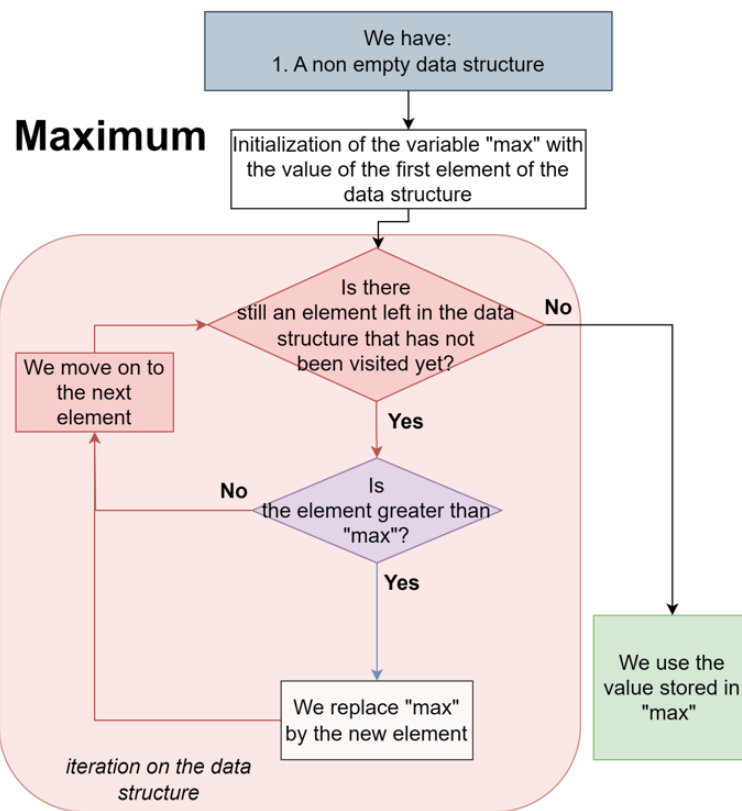


Figure 6.1: Flowchart of the Maximum Plan

6.2 The Plans

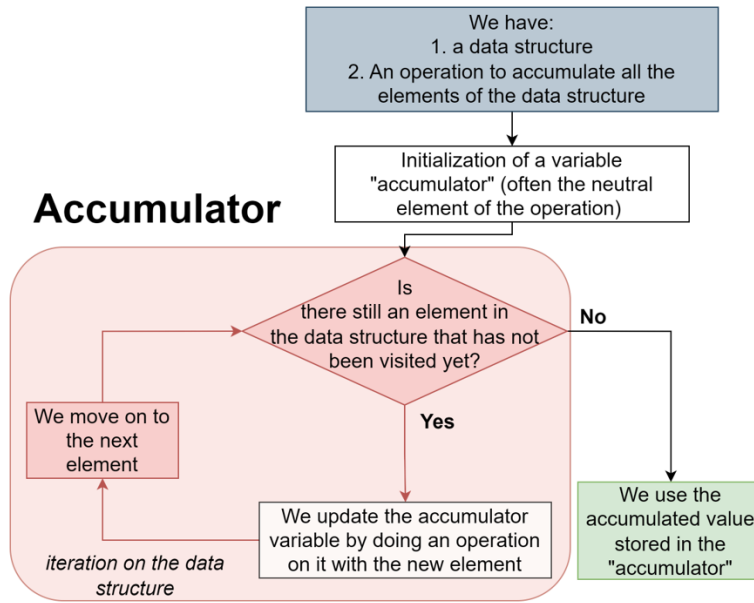
6.2.1 Designing the diagrams

For the visualisation of the different plans we created flowcharts paired with pseudo-codes. The flowcharts were designed to be the most conceptual possible. That way, one flowchart can represent the same plan in different implementations. As such, the flowchart of plan "*Maximum*" for example, can be applied to lists or dictionaries and can be implemented with a for loop or a while loop. This flowchart is visible on Figure 6.1.

The flowcharts are organized to represent a sequence of code execution. Rectangular boxes are used to represent an action. This can express for example the action of initializing or updating a value. As the prerequisites and the result are also considered actions, even if they are not part of the plan, they are also represented in a box. For example, after calculating the sum of the elements of list, an action has to be conducted, whether it is a `print()`, returning a result or yet another action.

The pseudo-codes on the other hand are designed to be the most specific possible to the certain plan, while staying applicable to a lot of different problems. This allows student to have a lot of scaffolding and allows them to understand how to use the plan in practice. For each flowchart, a lot of pseudo-codes are possible, as the pseudo-code depends on the data structure and choices of implementation. For clarity purpose, we mainly designed pseudo-codes with for loops, as they are more compact and easier to read. A lot of the pseudo-codes are designed with lists as well. That is because the first data structure that students learn about and in which all plans make sense are lists (in fact, the first data structure students really get acquainted with as such are strings, but the plan "*Maximum*" for example would not make much sense to them practically).

Since there are lots of ways to implement the different flowcharts, especially when considering every data structure, not every case has had its pseudo-code developed. We will discuss the main pseudo-codes used in practice, from which all the other pseudo-codes can be deduced. As the course is given in Python, naturally the pseudo-codes also use the Python syntax. In order to keep some coherence we will first present the pseudo-codes for each plan for lists and for loops. For a specific pattern we will then detail the difference between for and while loops. Finally we will discuss the difference in pseudo-code for each data structure.



(a) Flowchart of the Accumulator Plan

```

a_list = (some predefined list)
accumulator = some value

for element in a_list:
    accumulator = operation(accumulator, element)

return accumulator
  
```

(b) Pseudo-code of the Accumulator Plan for lists and a for loop

Figure 6.2: Diagrams for the Accumulator Plan

6.2.2 Accumulator

The Accumulator plan represents a pattern in which, as a data structure is traversed, a variable (the accumulator) is updated with every element of the data structure. This is for example the plan for calculating the sum of the elements of a list.

The flowchart of the Accumulator Plan is illustrated on Figure 6.2 (a). The prerequisites for this plan are the data structure and the operation with which we want to accumulate all the elements of the data structure. Typical operations are for example addition or multiplication. The plan begins with the initialization of the accumulator to the neutral of the operation. This can be zero in the case of an

addition, or one for a multiplication for example. After processing every element of the data structure, we use the value stored in the accumulator.

The pseudo-code for this plan is shown in Figure 6.2 (b). The particularity of this implementation is the way the operator is expressed. Since it can be a lot of different operations, we modeled it as a function using accumulator and element. In this plan we can see that the pseudo-codes are designed to be easily adaptable. The accumulator can for example be initialized at any value, and can be updated with every operation. This said, it is important that in this plan the accumulator is updated depending on the current element and not a constant. We made that distinction because in the way of thinking and in the application on specific problems this was a fundamental difference.

6.2.3 Accumulator With Condition

A very similar plan is the Accumulator With Condition Plan. It is virtually the same except a condition is added in the iteration. This allows the student to filter out some of the elements and to not include them in the calculation of the accumulator. An example of this plan could be to calculate the sum of all even elements of a list. One of the reasons this plan was added as a whole separate plan and not a contraction of two plans was to bring to light a comparison with other patterns, which we will discuss further later.

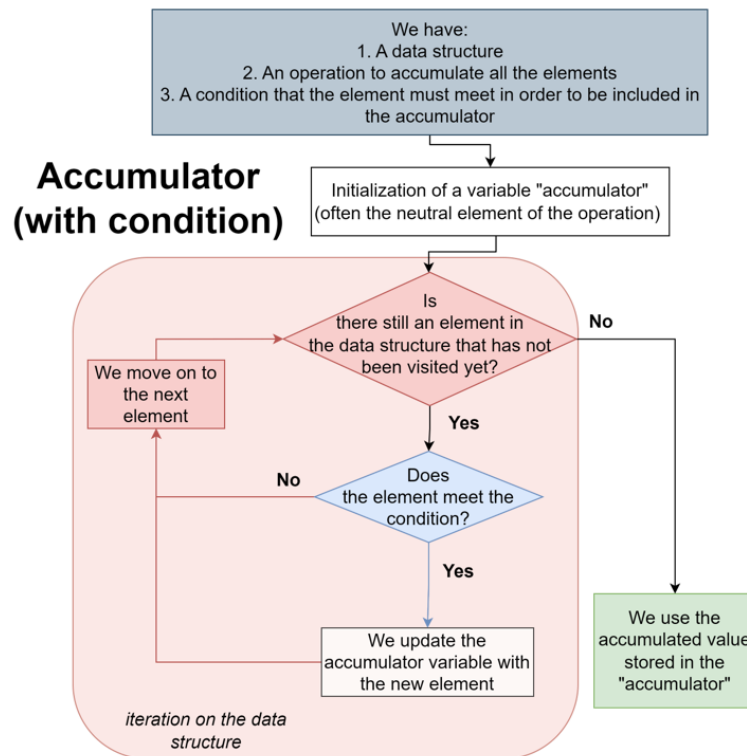
Like mentioned previously, the flowchart of Accumulator With Condition Plan, on Figure 6.3 (a) is largely the same as the Accumulator Plan. A prerequisite and a blue diamond are added to respectively represent and express the filtering condition.

The pseudo-code of the Accumulator With Condition Plan (Figure 6.3 (b)), adds a condition in the loop. Since the syntax of the condition depends on the condition itself, we expressed it as a function depending on the element. The rest is similar as the previous plan.

6.2.4 Filter

The Filter Plan represents a plan that loops over a data structure to filter out some elements based on a condition. The resulting elements are stored in another data structure. This plan can for example be used to filter out all negative elements from a dictionary. The particularity of this flowchart, as illustrated on Figure 6.4 (a), is that an empty data structure is created. This structure will be used to store the resulting elements of the filter. The pseudo-code for the Filter (on Figure 6.4

(b)) initializes an empty list and appends elements to it. Since it is specific to the data structure and not directly to the problem in itself, we can be quite specific and express this literally.



(a) Flowchart of the Accumulator With Condition Plan

```
a_list = (some predefined list)
```

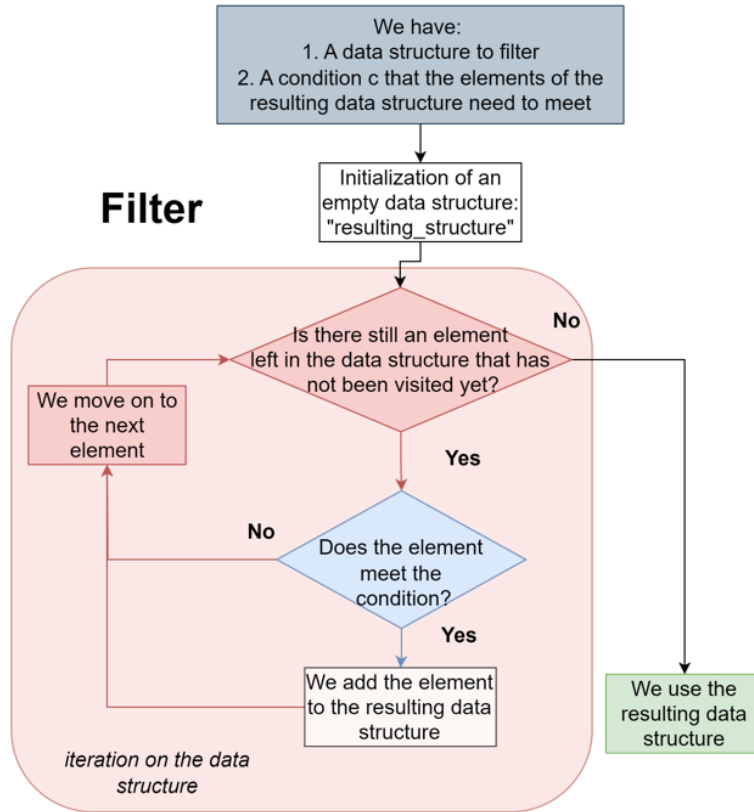
```
accumulator = some value
```

```
for element in a_list:
    if condition(element):
        accumulator = operation(accumulator, element)
```

```
return accumulator
```

(b) Pseudo-code of the Accumulator With Condition Plan for lists and a for loop

Figure 6.3: Diagrams for the Accumulator With Condition Plan



(a) Flowchart of the Filter Plan

```

a_list = (some predefined list)
resulting_list = []
for element in a_list:
    if condition(element):
        resulting_list.append(element)
return resulting_list

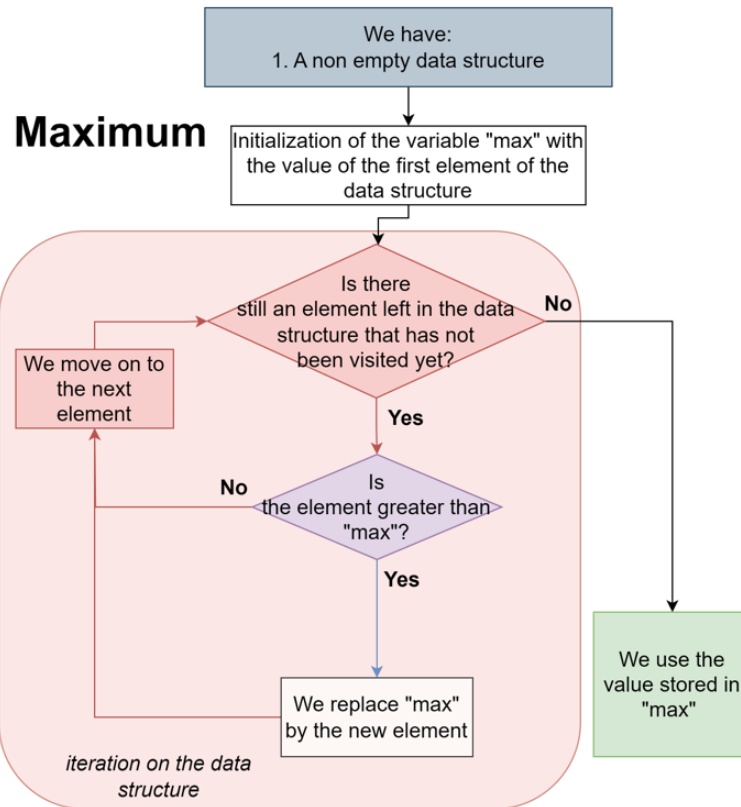
```

(b) Pseudo-code of the Filter Plan for lists and a for loop

Figure 6.4: Diagrams for the Filter Plan

6.2.5 Maximum

A more specific plan is the Maximum plan, in which the maximum value of a data structure is searched for. It is implemented in such a way that the maximum value is searched for, but can be easily adapted so that, for example, the index of the



(a) Flowchart of the Maximum Plan

```

a_list = (some predefined list)

maximum = a_list[0]

for element in a_list:
    if element > maximum:
        maximum = element

return maximum
  
```

(b) Pseudo-code of the Maximum Plan for lists
and a for loop

Figure 6.5: Diagrams for the Maximum Plan

maximum value of a list is found. Its flowchart, on Figure 6.5 (a), includes an initialization of an initial maximum, the first element of the data structure. The

condition here is specific, as well as the action associated (namely replacing the previous maximum by the new maximum).

We did not detail every exception that might occur. For example, it does not model the case where the data structure is not empty, or when the elements are not comparable. The objective was to keep this flowchart the most simple to show the pattern without confusing the student with corner cases.

The Maximum plan, whose pseudo-code is depicted on Figure 6.5 (b), is the most specific pattern and thus has the least flexibility. The fact it is so specific allows the condition to be explicit, and directly applicable for students. The advantage of such an explicit plan is that it allows students to see even more concretely how a plan is coded.

6.2.6 Verification

The Verification Plan represents a plan where a data structure's validity is verified. If one of the elements does not meet the validation condition, the loop is exited and False is returned.

Its flowchart, on Figure 6.6 (a), is different than the other plans structure-wise. When finding an element that does not meet the condition we exit the loop. This is represented with the light red color indicating the scope of the loop. Here two outcomes are possible, resulting in two green boxes.

The pseudo-code of the Verification pattern, on Figure 6.6 (b), is broadly similar to the other pseudo-codes, with the major difference that it is possible to exit the loop while it has not yet iterated over all elements. This difference is more important conceptually in the flowchart than practically in the pseudo-code.

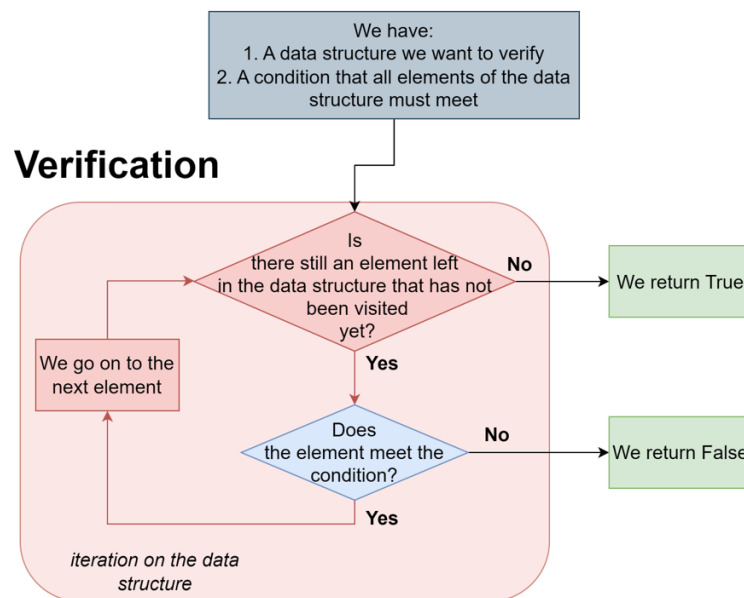
6.2.7 Counter

Finally, the last plan "Counter" is used to count how many elements of a data structure meet a certain condition. This could for example be used to count how many numbers of a list are prime numbers.

Its flowchart, on Figure 6.7 (a) is very similar in structure to the Accumulator With Condition Plan. Here we have a variable *counter* we increment every time an element complies with the condition. There are two main differences with the

Accumulator With Condition Plan: the increment here is constant and does not depend on the element itself. We chose not to merge these two plans because of the amount of times the Counter plan is applicable on exercises on its own. Merging the Counter and Accumulator With Condition plans would have resulted in too large a category.

Finally, for the pseudo-code of the Counter plan on Figure 6.7 (b), we decided to remain very explicit and set the start of the counter to zero and the increment to one. We decided to do this because it will be the case in most of the exercises the students will have to solve. A counter with a constant increment other than one is not very common and would confuse students as to when to use this plan.



(a) Flowchart of the Verification Plan

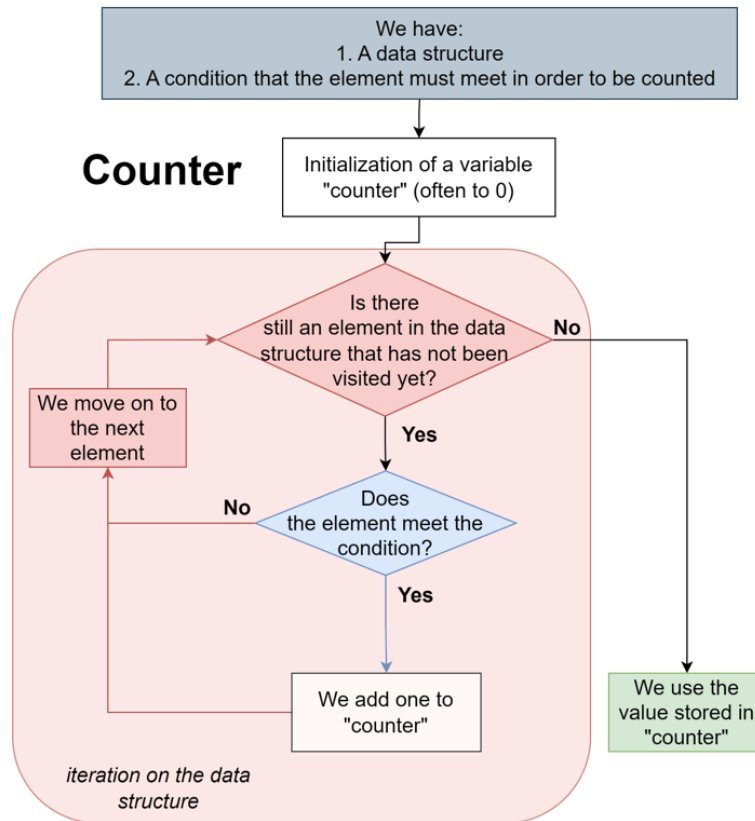
```

a_list = (some predefined list)
for element in a_list:
    if not condition(element):
        return False
return True

```

(b) Pseudo-code of the Verification Plan for lists and a for loop

Figure 6.6: Diagrams for the Verification Plan



(a) Flowchart of the Counter Plan

```

a_list = (some predefined list)
counter = 0
for element in a_list:
    if condition(element):
        counter = counter + 1
return counter

```

(b) Pseudo-code of the Counter Plan for lists and a for loop

Figure 6.7: Diagrams for the Counter Plan

6.2.8 Other implementations of pseudo-codes

The previous pseudo-codes used for loops as they are generally more compact. An example of pseudo-code with while loops is illustrated on Figure 6.8 (a). To show the plans on different data structures, we chose to illustrate this with the Filter plan. This pattern was chosen because it is the only pattern including an

initialization of the data structure. Moreover, it also shows how to add elements to it and it is possible to find examples of meaningful applications for each of the data structure seen in the course. Pseudo-codes for the different data structure are depicted on Figure 6.8 (b), (c) & (d).

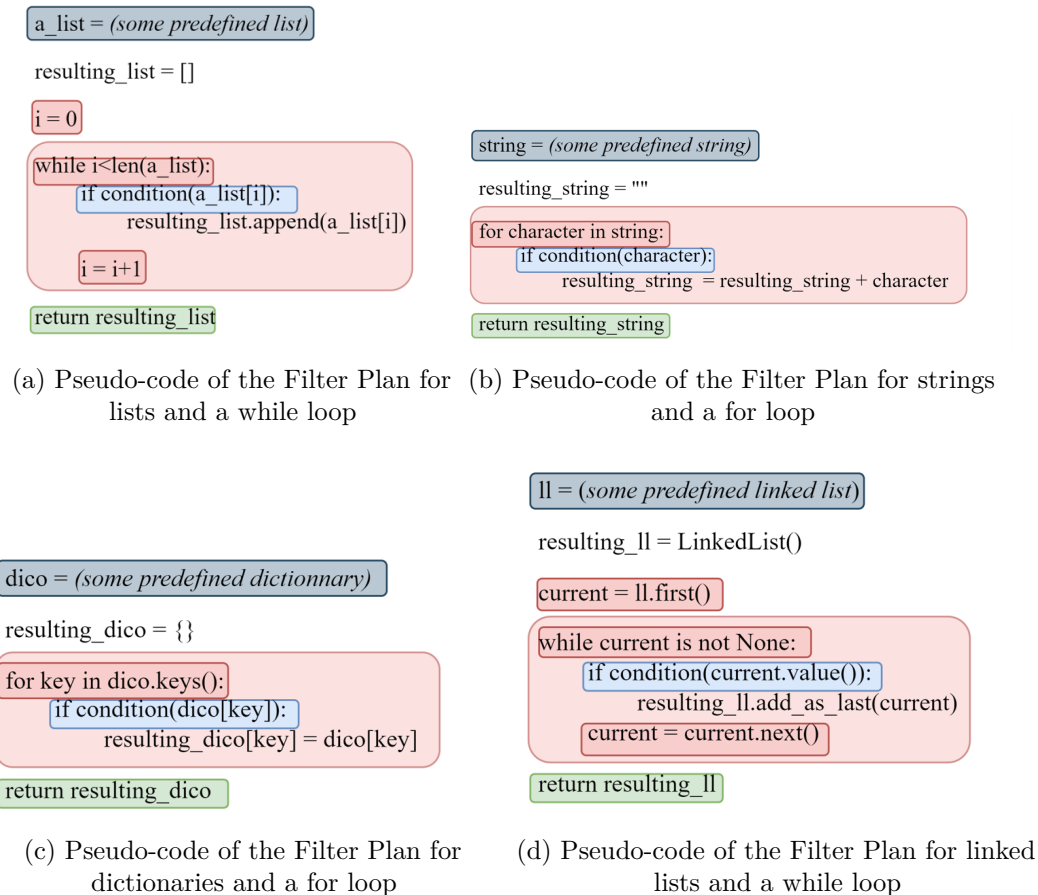


Figure 6.8: Different implementations of the pseudo-code for the Filter Plan

Our three axis of modification are thus the type of loop, the type of data structure and the plans. The associated constants for showing the different cases are respectively for loops (except for Linked Lists), lists and the "Filter" pattern.

6.2.9 Color Code

A same color code is applied to both the flowcharts and the pseudo-codes, so that the link between the two are explicit and clear.

The prerequisites are in a grey box. They are often the given of a problem, or a data structure that needs to be changed. For example in *"Filter"*, the prerequisites are the condition of filtering and the data structure that needs to be filtered.

The red is used to illustrate iterations. Darker red is used to represent the code responsible for this iteration, and light red is used to show its scope. The darker red encompasses all code necessary to create a loop. That means that in the case of a while loop for the plan *"Accumulator"*, the darker red is also used for the code initializing and updating the index *i* for example, as shown on Figure 6.9.

```
a_list = (some predefined list)
accumulator = some value
i = 0
while i < len(a_list):
    accumulator = operation(accumulator, a_list[i])
    i = i + 1
return accumulator
```

Figure 6.9: Pseudo-code for the Accumulator pattern with a while loop

Green is used to show the result. This way, the student understands the goal of the operation. For example in the case of the *"Accumulator"*, we want to use the result of what is stored in the variable *accumulator*.

Finally, we used two colors for conditions: blue and purple. Blue is used for conditions expressing a belonging to a certain group. This is for example the case in the plan *"Filter"*. The condition expresses if the element of the data structure belongs to the group that is going in the resulting data structure or not. A practical example of such condition could be "all numbers that are odd".

Purple is only used in the *"Maximum"* pattern, as it is the only pattern without a condition of "belonging". Since the pattern in itself is quite specific, this color as well. It specifically represents the condition verifying whether the element is greater than the current maximum or not.

6.2.10 Similarities in Plans

The main pattern from which three other patterns are derived is the "Accumulator with condition" pattern. Indeed, the most obvious example is the "Accumulator" pattern, which simply removes the condition. "Counter" also represents an accumulation, but it is constant and does not depend on the element of the data structure. Finally, the resulting data structure in "Filter" can also be interpreted as an accumulator, and the operation then becomes, for example, *append()* in the case of lists. This is also explained in the theoretical syllabus.

6.3 Insertion in the course

Students are first introduced to these diagrams in the theory syllabus [10]. In the associated INGINious exercises, those plans are reminded explicitly. As the students become more experienced, these hints become less frequent or more implicit. How these diagrams are presented and how those hints are put in place will be explained in this section.

6.3.1 Theory syllabus

The first time we mention these plans in the syllabus is when students discover the principle of iteration. They are presented two problems: the sum of the numbers from one to ten, and the subtraction of the numbers from one to hundred. The goal is to show them that different problems can have the same structure in their solution. Here the example is quite obvious since we wanted to start with an easy plan. We display two possible solutions for the problems and then proceed to extract the pseudo-code of the Accumulator pattern out of it. Those specific implementations are shown on Figure 6.10. Only after showing the pseudo-code we show the associated flowchart. After this, the opposite will be done: first the flowchart and then a possible pseudo-code. The first time we put the focus on understanding the goal of the plans and on showing how it could be useful. But for the other plans we do the opposite to encourage the student to think about the general structure of his or her solution before coding.

The diagrams we first show them of the Accumulator pattern have been slightly modified to fit their level of understanding. Firstly, since they have not yet seen functions, they don't know what *return* statements are. These are therefore replaced by *print()* instructions. Nor do they know exactly what data structures are, since they've only briefly seen lists and strings used, but not as such. We modified both the flowchart and pseudo-codes to apply them to "sequences of numbers" (generated

Problem 1

```
sequence= range(1,10)
```

```
accumulator = 0
```

```
for element in sequence:
    accumulator = accumulator + element
```

```
print(accumulator)
```

(a) Solution of the first Problem

Problem 2

```
sequence= range(1,100)
```

```
accumulator = 0
```

```
for element in sequence:
    accumulator = accumulator - element
```

```
print(accumulator)
```

(b) Solution of the second Problem

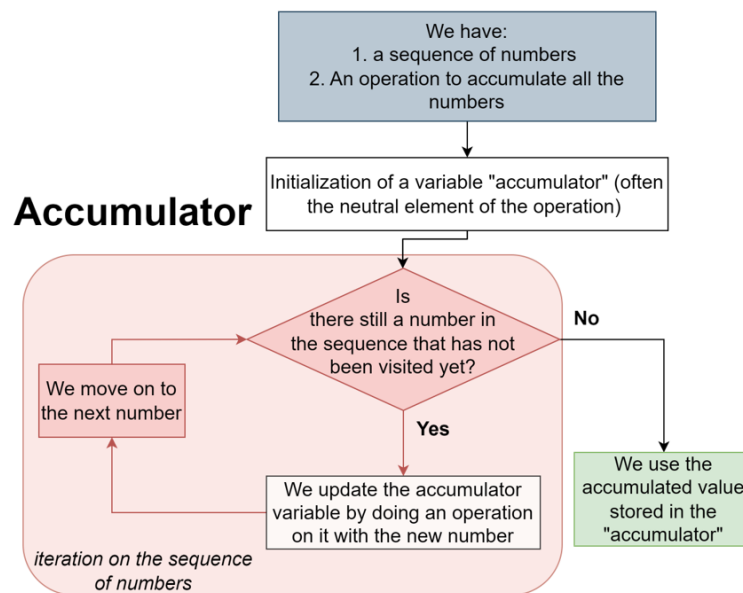
```
sequence = range(begin,end,step)
```

```
accumulator = some value
```

```
for number in sequence:
    accumulator = operation(accumulator, number)
```

```
print(accumulator)
```

(c) Pseudo-code for the problems



(d) Flowchart of the Accumulator pattern for a sequence of numbers

Figure 6.10: Introduction to the Plans in the theory syllabus

with the *range()* function). This is visible for the Accumulator plan on Figure 6.10 too. After this we introduce a second pattern: the Counter plan, with the same tweaks (a *print()* statement instead of a *return* and a sequence of numbers instead of the word "data structure").

At the end of the theory about strings we introduce the Verification Plan. Students have now already seen what functions are so the *print()* statements can become *return* statements. However, they still have not explicitly seen a lot of data structures, so the flowchart has been adapted to strings instead of staying very general.

It is only when discovering lists that we introduce the rest of the patterns. We first show all the previous patterns with the flowchart generalized to data structures as well as all the pseudo-codes with *return* statements and lists. We then proceed to explain the rest of the plans, namely Accumulator With Condition, Filter and Maximum. Every time the associated pseudo-code is shown for lists and sometimes illustrated with a practical example. We kept this quite short as students should already understand the concept of plans at that point and have already seen half of the plans before.

Every time a new data structure is introduced in the theory syllabus, a pseudo-code of the pattern "Filter" is created, showing students how to modify the plans to fit the specific data structure. This will be the case when introducing dictionaries and linked lists, as illustrated on Figure 6.11. The advantage of the "Filter" pattern is that it allows us to show almost all possible operations on a data structure. It includes initializing a structure, adding elements and accessing elements of the structure.

6.3.2 INGIInious exercises

The first step was to identify the different plans in the exercises. To give an idea on the frequency of these patterns in the INGIInious exercises, figure 6.12 shows a general timeline of the different patterns in the exercises up to week four. Out of the 43 coding exercises of week one to four, 18 implement one or more plans. To help students translate theory into practice, several solutions have been thought of. They vary in degree of help but all use the patterns in some way. All those helps are directly displayed with the exercise statement. We will discuss them starting from a lot of guidance to a small help.

Patterns applied to dictionaries

All the patterns we have already seen can be applied to dictionaries. To illustrate this, let us take the **"Filter"** pattern. The pseudo-code with for loops then becomes:

```
dico = (some predefined dictionary)

resulting_dico = {}

for key in dico.keys():
    if condition(dico[key]):
        resulting_dico[key] = dico[key]

return resulting_dico
```

The modification of the other plans to fit dictionaries is very similar.

(a) Introduction of the Filter plan for dictionaries

Patterns for linked lists

Like every other data structure, the plans we have identified for lists can also be adapted to linked lists. As an example, let's take yet again the example of the **"Filter"** pattern. The pseudo-code becomes this:

```
ll = (some predefined linked list)

resulting_ll = LinkedList()

current = ll.first()

while current is not None:
    if condition(current.value()):
        resulting_ll.add_as_last(current)
    current = current.next()

return resulting_ll
```

Notice that it is necessary to add the new nodes as last, and not as the head like the normal method "add(self, cargo)" does. Otherwise, the resulting linked list would be reversed. A possible code for add_as_last() in the LinkedList class could be:

```
def add_as_last(self, cargo):
    node = Node(cargo, None) #create the node that is going to be added
    if self.__head == None: #If the linked list is empty
        self.__head = node
        return

    current = self.__head
    while current.next() is not None: #While we have not reached the last node
        current = current.next()
    current.set_next(node) #The next node of the (previously) last node becomes the created node
```

(b) Introduction of the Filter plan for Linked Lists

Figure 6.11: Introduction of different data structures in the Theory Syllabus

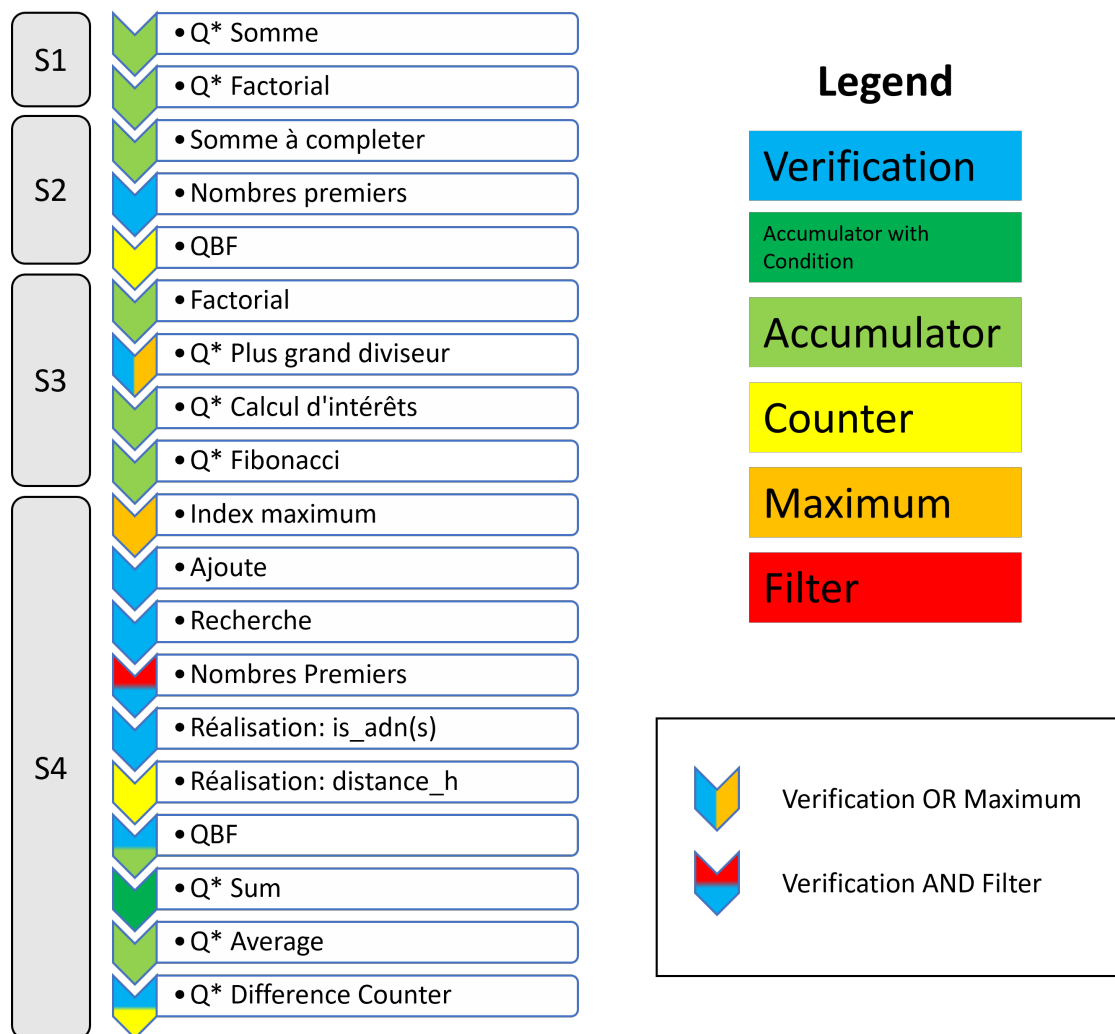


Figure 6.12: Plans occurring in the INGINious exercises until Week 4

Scaffolding

One way to show students how to apply those patterns is by implementing their structure in the exercises. For those exercises, the pseudo-code is already visible to the students, with blanks they have to fill in. This is for example the case with the exercise "Q* Somme", as showed on Figure 6.13. In this optional exercise of week 1, students are asked to store the sum of the numbers from 1 to x in a variable result. This is a typical example of the Accumulator pattern, and we can thus provide the parts of the pseudo-code that are fixed for this particular pattern. However, this is a lot of support, so it will not be implemented for a lot of exercises. The main goal for this help is show students how to take advantage of these plans.

Session 1: Q* Somme [Edited]



In programming, one of the first exercise asked for practice is to calculate the sum of the n first positive integers. It is to illustrate the capacity to repeat a process easily and with a few lines of computers.

To do so, use a `for` loop with a range.

Note: The code fragment we give you is the basis structure of the Accumulator pattern, described in the theory syllabus (<https://drive.google.com/drive/folders/1wk95Y2DQRDorIsxd8h2PoMGIGHWnbNOY?usp=sharing>).

Implémentation

```
x = ... #le nombre
result = ... #enregistre dans la variable la somme des c premiers entiers positifs
```

```
1 result = _
2 for ____ in ____:
3     result = result _ ____
```

Figure 6.13: Caption of the Q* Somme exercise with scaffolding

Session 1: Q* Factorial [Edited]



In mathematics, the factorial of a strictly positive integer x , denoted as $x!$ is the product of all strictly positive integers less than or equal to x . The factorial of 0 is defined as 1. Calculate the factorial of an integer x and store the computed value in the variable `result`.

Examples:

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $0! = 1$

Hint: If you encounter some problems knowing how to proceed, the "Accumulator" pattern can help you get started. Notice the similarity with the exercise Q* Somme. (Link to the theoretic syllabus extract:

Figure 6.14: Caption of the Q* Factorial exercise with a hint directly based on the plans and a hint based on another exercise

Hints directly based on the plans

Another type of help developed in the exercises are hints describing what plan that particular exercise uses. That way, students who are having trouble with starting the exercise, or don't know how to proceed can base their code on the pseudo-codes and adapt them to the problem at hand. An example of this type of hint is available on Figure 6.14, which depicts the exercise Q* Factorial, of Week 1. This type of hint will be the most implemented in the INGINIOUS exercises, as it does not give too much detail and is applicable on a lot of these exercises.

Hints based on other exercises

The last type of hint consists of making links between exercises. The idea is to reuse the same thinking process and parts of the code of a previous exercise to solve a new one. This encourages the student to see the links between the topics and thus deepen his comprehension of the subject. This is illustrated at the Figure 6.14, with the exercise Q* Factorial, where students have to store the factorial of number x in the variable result. In this example it is used in combination with a hint based on a pattern as it is one of the earlier weeks, but it could also be used as only hint. This is for example the case in the exercise "Q* Calcul d'intérêts", as presented on Figure 6.15. However, although they are useful, opportunities to implement those types of hints are quite rare. Indeed, students already have a lot of exercises to do, and adding too many exercises can be overwhelming. As such, there is a need to carefully select the exercises and avoid redundancy. Exercises are thus varied and a direct transposition of a solution of one problem onto another problem is not often possible.

Session 3: Q* Calcul d'intérêts [Edited]



Suppose you now have **base** € on your savings account.

If you leave it **x** years on that account at a fixed cumulative interest rate of **y** %, what will be the balance on your account after these **x** years?

Return that amount.

Hint:

This exercise is similar to the exercises *Somme à Compléter* of Week 2 and *Factorial* of Week 3.

Figure 6.15: Caption of the Q* Calcul d'intérêts exercise with hints based on other exercises

Chapter 7

Validation

Due to a timing issue, a validation with students will not be possible. Indeed, the CS1 course is given during the first semester, from September to January, while this thesis will be handed in in June. However, for a future potential validation, we will describe how a student validation could be done. In addition, we did a validation with tutors which we will describe here as well.

7.1 Potential validation with Students

For the student validation we want to investigate if the integration of our patterns promotes the transfer of knowledge from one exercise to another. We will measure this by taking as metric the time necessary to solve an exercise after succeeding a similar one before.

In order to achieve this, we will take two specific sets of two exercises. The first two exercises are "Q* Somme" and "Q* Factorial" which we already presented previously. Their description is available on Figures 7.1 and 7.2. Both are based on the "Accumulator" plan and are very similar. The second set are the two exercises from Week 2 and 4: "Is Prime" and "Ajoute". The first exercise will be an adaptation from the current exercise of Week 2, where the student has to return whether a number is prime or not. In the second exercise, the student is asked to add a number to a list, only if the list does not yet contain this number.

For the validation, students will be divided into three groups. First is the control group, who will have no hints or scaffolding for any of the exercises. The second group will only have hints based on the plans for the first exercise of each set. The third group will only have scaffolding for the first exercise of each set. By measuring the time it takes for students to succeed the second exercise of each

Session 1: Q* Somme [Edited]



In programming, one of the first exercise asked for practice is to calculate the sum of the n first positive integers. It is to illustrate the capacity to repeat a process easily and with a few lines of computers.

To do so, use a `for` loop with a range.

Note: The code fragment we give you is the basis structure of the Accumulator pattern, described in the theory syllabus (<https://drive.google.com/drive/folders/1wk95Y2DQRDorIsxd8h2PoMGIGHWnbNOY?usp=sharing>).

Implémentation

```
x = ... #le nombre
result = ... #enregistre dans la variable la somme des c premiers entiers positifs
```

```
1 result = _
2 for ____ in ____:
3     result = result _ ____
```

Figure 7.1: Caption of the Q* Somme exercise with scaffolding

Session 1: Q* Factorial [Edited]



In mathematics, the factorial of a strictly positive integer x , denoted as $x!$ is the product of all strictly positive integers less than or equal to x . The factorial of 0 is defined as 1. Calculate the factorial of an integer x and store the computed value in the variable `result`.

Examples:

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $0! = 1$

Hint: If you encounter some problems knowing how to proceed, the "Accumulator" pattern can help you get started. Notice the similarity with the exercise Q* Somme. (Link to the theoretic syllabus extract:

Figure 7.2: Caption of the Q* Factorial exercise with a hint directly based on the plans and a hint based on another exercise

set we can compare each group and have an indicator of their effectiveness for knowledge translation.

Ideally, for this to be representative, all students should participate and not only volunteers. Indeed, there could be a bias since people with more confidence in their programming abilities are more likely to volunteer for programming experiences, and our help targets students experiencing difficulties with programming. However, from a legal point of view it is probably not possible to perform such validation without permission, as all students should be on an equal footing and have equal opportunities to succeed the course.

Another way of assessing the effectiveness of the different formats of help is to ask

students what they think has helped them most in solving the exercises throughout the year. In this scenario, different types of hints or scaffolding would have been given throughout the year in all kinds of exercises and at the end of it, students would get the possibility to give their opinion on what helped them the most. This would allow to also have feedback about hints related to previous exercises, which is a type of hint we could not include in the first scenario. There would also be no problems from a legal point of view. The downside of this however, is that there may be a bias. We evaluate the types of help on the students' perception of efficiency but not on their actual efficiency to transfer knowledge.

7.2 Validation with Tutors and Assistants

7.2.1 Method

For the second validation, the theory behind the plans was presented to the tutors, followed by a presentation of the diagrams and their practical implementation using two scenarios. The tutors were then asked to fill in a survey.

We first introduced the concept of patterns and their corresponding diagrams to tutors the same way they would be presented to students. Since they don't have access to the theory syllabus we modified, we compiled the added parts of it into a document that the tutors could access. In order not to make the task too time-consuming, we have added only the explanations of the plans necessary for the resolution of the survey, namely the Accumulator, Verification and Filter plans, leaving aside the explanations relating to the other plans.

After reading the theory behind the models and diagrams, the tutors followed two scenarios, each consisting of two exercises that they had to solve. In the first set of exercises they were asked to solve two optional exercises of the first week: the exercise Q* Somme and the exercise Q* Factorial. The scaffolding of the plan "Accumulator" was given for the first exercise, and for Q* Factorial two hints were given: a hint regarding the pattern of this exercise and one regarding the link with Q* Somme (both exercises are implementing the "Accumulator" plan). Immediately after they solve this first set, we ask them the following question: *"As a tutor, to what extent do you think that solving the first exercise helps students to solve the second?"*, leaving them the possibility to choose an answer between "not at all", "rather not", "neutral", "quite a lot" and "a lot".

In the second scenario, they were asked to solve the exercises "Ajoute", with a hint regarding the plan the exercise uses (namely Verification), and "Prime Numbers".

For the second exercise, a breakdown into two sub-problems was suggested, with for each sub-problem a hint on which plan one can use to solve it: Verification and Filter. Similarly, tutors were asked to what extent solving the first exercise can help the student solve the sub-problem "is_prime", of the second exercise, as Verification was the plan behind both those (sub-)problems. Finally, tutors were also asked which of the three types of help (scaffolding, hints on the plan and hints on the link with a previous exercise) seemed the most relevant to them.

These scenarios were presented for various reasons. Firstly they were a good opportunity to show how the diagrams could be used practically. In these four exercises, the three ways of implementing those diagrams to help students have been also demonstrated. The chosen exercises of a same set had the same pattern and could thus illustrate the similarities in exercises.

After presenting both scenarios, we asked the tutors to answer various questions, each time on a scale from one to five with one being "not at all" and five being "a lot". Two means "rather not", three means "neutral" and four means "quite a lot". These questions concern the clarity and usefulness of the diagrams and pseudo-codes, their ease of integration into a tutored session or the ease of understanding the concepts underlying these diagrams and the links between them. We also asked them whether the two main objectives of these diagrams were being met, namely to create links between exercises and to help the students know how to get started. Finally, they were also given the opportunity to make comments about things that could be improved, or simply more general remarks.

7.2.2 Results

We received a total of seven responses to our survey. For a lot of the questions, tutors had to evaluate a specific criterion on a scale from one to five. The responses of these questions (which form the large majority of the questions regarding the whole survey) are displayed in a table in Figure 7.3. For each question, the number of people who voted for the corresponding number is displayed, along with the total average in the last column.

For the remaining three questions, we will detail the results here. To the question *"During the exercises, different formats of help are offered. Which one(s) do you think are relevant?"*, five tutors answered that scaffolding would be relevant, four answered that a reference to a certain pattern would be relevant and everyone found that a reference to another exercise would be relevant. The two last questions are more general questions: *"Do you have any comments or suggestions for improvements or changes?"* and *"Any other comments or remarks?"*. As answer to

these questions, many tutors bring perspectives on the course, more than on the diagrams in themselves. For example, one tutor points out that students do not read the syllabus very much, mainly because it is in English. Another tutor also commented that the exercise statements were not always very clear and that they needed to be structured more. However, a relevant point that emerged from these questions about diagrams is that *"the abstraction between pseudo-code or diagrams and real Python code is difficult for the students. [...] They understand the logic behind diagrams (they understand the algorithmic basis) but they get very stuck on syntax and the fact that they don't have a pre-made code structure. There's a sort of blank page syndrome; starting from scratch is difficult for them"*. Finally, another tutor also mentions that it's not very pleasant to have the text coming out of the diamonds in the flowcharts. This has been corrected in the meantime.

7.2.3 Analysis/discussion

The discrepancy in results between the first two questions is not surprising. Indeed, the first set of exercises contain two of the most similar INGINIOUS exercises. They both use the simplest plan "Accumulator" as it is. The exercises of the second set however, were not so overtly similar. Students need to take a step further after completing the first exercise in order to succeed the second. So the lower score of 3.14 for the second as opposed to 4.14 for the first is not unexpected. However, the rather low score of 3.14 is not very promising, as one of the objectives of the diagrams was precisely to establish links between exercises that were not similar at first sight.

Similarly, for the third question, the comparatively low score is understandable, since the link between the patterns are not very explicit for the exercises of the second set. The structure is the same but the thinking behind it is very different. The verification pattern in the case of 'Ajoute' for example, is applicable as the students first have to verify whether the list already contains the value or not, which is an clear implementation of the Verification pattern with some tweaks. However, for the `is_prime()` sub-problem, where students have to define a function that returns whether a number is prime or not, it can seem less obvious. The suggested way of implementing this is by verifying whether the number is divisible by one of the numbers smaller than him. If it does, `is_prime` has to return False. Otherwise it is True.

Questions 4 & 5 deal with how well our two objectives were achieved. Our second objective, which was to create a starting point for students to code, seems to have been met according to the tutors. This is an encouraging result, as it enables students to focus more on the process, by giving them steps to start with. Opinions on the first objective, however, seem to be more divided. This can be also explained

		1	2	3	4	5	Average [1;5]
1	As a tutor, to what extent do you think that solving the first exercise helps students to solve the second?	0	0	1	4	2	4.14
2	As a tutor, to what extent do you think that solving the first exercise helps students to solve the is_prime(number) sub-problem in the second?	0	2	3	1	1	3.14
3	As a tutor, to what extent do you think that the flowcharts and plans help students to solve the exercise, such as the "Verification" plan in Ajoute or "Filter" in the second sub-problem of Prime Numbers?	0	0	3	3	1	3.71
4	One of the aims of diagrams/plans (such as "Accumulator" or "Verification") is to create links between exercises. To what extent do you think this objective has been met?	0	1	2	2	2	3.71
5	One of the aims of diagrams/plans (such as "Accumulator" or "Verification") is to help students know how to get started. To what extent do you think this objective has been met?	0	0	0	3	4	4.57
6	On a scale of 1 to 5, how would you rate the usefulness of the flowcharts?	0	0	1	4	2	4.14
7	On a scale of 1 to 5, how would you rate the clarity of the flowcharts?	0	1	0	3	3	4.14
8	On a scale of 1 to 5, how would you rate the usefulness of the pseudo-codes?	0	1	0	3	3	4.14
9	On a scale of 1 to 5, how would you rate the clarity of the pseudo-codes?	0	1	1	1	4	4.14
10	On a scale of 1 to 5, how would you rate the ease with which diagrams (flowcharts and pseudo-codes) can be integrated into sessions, for example?	0	1	2	4	0	3.43
11	On a scale of 1 to 5, how would you rate how easy it is to understand the concepts behind the diagrams?	0	2	2	3	0	3.14
12	On a scale of 1 to 5, how would you rate the ease of understanding the links between the plans?	0	3	1	3	0	3

Figure 7.3: Result of the Tutors' Validation for a part of the questions

by the difference in both sets, as it is easier to establish a link between the exercises of scenario 1 than it is in scenario 2. Another way this can be explained is that there are not a lot of examples on how to practically implement those plans in real programming problems. There are several pseudo-codes for each plan, but this can still be too abstract for beginners.

The scores regarding the clarity and utility of both the flowcharts and the pseudo-codes are quite homogeneous, with a score of 4,14. This is encouraging, but it also

shows that there is room for improvement. Possible ways of improving could be to add more concrete examples, with more explicit links between flowcharts and pseudo-codes on one side, and between pseudo-codes and solutions to real concrete problems on the other.

The three last questions deal with how easy those diagrams can be integrated and understood, on their own or together. This appears to be more difficult, as the scores are all below 4. One thing that should be noted, is that in the syllabus excerpt we did not include the paragraph explaining how the plans are similar to each other. This score is thus only based on how the diagrams were designed to have this uniformity intrinsically. However, these scores clearly indicate the need to make the concepts and links between plans and diagrams more explicit.

Finally, as shown by the answers to the question on different help formats, it appears that only four tutors would find a reference to a certain pattern relevant and would rather praise the approach of referring directly to other exercises. As we will discuss in the conclusion, this is unfortunately more difficult to implement as there are not a lot of exercises which share a clear link like it was the case in the first scenario. There are however more cases like the second scenario, but as it also showed in the tutors' validation, the link is less obvious and could confuse the students.

One peculiarity that emerges from the validation is the following: the tutors agree that the diagrams are not very effective in showing links between exercises, but when asked what types of hint they think are most relevant, 100% of the answers include a hint based on a previous exercise. The logical conclusion we can draw from this is that it may be better to hide these diagrams from the students and only use them as tools to identify similarities between exercises, enabling hints to be developed for certain exercises. The downside of this is that there are not a lot of set of exercises for which this could be implemented. Many exercises share the same patterns, but without the knowledge of those patterns it is difficult to see why they are similar and this risks confusing the students even more.

Another major disadvantage of patterns is that they are very limited. The further the students progress in programming, the less the patterns are applicable as they are. They are often part of a larger structure. The use of patterns as starting point of the coding process becomes more difficult, as they need to be adapted and combined. Their use to establish links also becomes less straightforward, as exercises may use the same patterns but applied in fundamentally different ways. Establishing links between those kinds of exercises thus becomes harder and may

not be very effective.

In general, the opinions are quite positive: the scores are generally around 4. Overall, the tutors fairly agree that these diagrams can be useful, especially to help students get started. This is also confirmed by the comment one tutor made about how students can have difficulties starting from scratch. However, the links between the exercises and the plans were not always clear, as can be seen from the answers to questions 3, 4 and 12. Overall, the diagrams are useful and clear, but there is still room for improvement. On the other hand, the integration with the course and the link with the course concepts are not yet clear enough and need to be improved.

7.2.4 Threats to validity

Since we did not get a lot of responses from the tutors we had to ask the assistants to answer the survey too in order to get a more meaningful feedback. In total, three assistants and four tutors have responded. Tutors are generally closer to the students as they lead the tutored sessions, whereas assistants act as a link between students and the teachers and generally do not participate in these tutored sessions. However, they answer a lot of students' questions and sometimes replace tutors during these sessions. They thus still have a good idea where the students' difficulties lie.

However, even with the assistants' help, we only received seven responses. As there are around thirty tutors and assistants in total, this represents only around a quarter of all tutors and assistants. Moreover, the responses are usually not very homogeneous, as there are often three to four different responses for a same question. The conclusions reached should therefore be taken with a grain of salt.

Another factor that may explain the heterogeneity of responses is the fact that the examples of the two scenarios are quite different and do not represent the plans as clearly as one another. Retrospectively, it would have been preferable to split certain questions to obtain an answer for each series.

Chapter 8

Final Discussion

8.1 Future Work

While searching for solutions to diversify the type of feedback, different options came to mind. These paths were not chosen for various reasons that we will detail in this section, however they could be worth to develop. As the reflection was turned towards the exercises, a lot of these ideas relate directly to them. The feedback ideas are structured according to the timing: before, during or after the student has done the exercise.

8.1.1 Before the exercise

When analyzing the feedback, we also analyzed the exercises themselves and examined the errors made by the students. The root of a lot of the mistakes were a bad comprehension of the task requirements. Sometimes, when submitting their code, an adequate feedback is given, as for example "The function: {}, does not call the function rho" (Q* Equations du second degré, Week 3). But more often than not, students don't understand what the problem is. It is due to two types of problems. They either don't understand correctly the problem and the goal of the task, or they don't understand the format they are supposed to write it in. Sometimes they have to put a certain value in a variable, other times they have to create a function for example.

For the first problem, a possible solution could be to make the exercise statements clearer. A way to do that would be to add examples of use cases of their solution. In their article Ott et al. [6], particularly stress the importance of a clear task statement. This would admittedly not increase the diversity in feedback types as it would be classified as Hints on Task Requirement by Keuning et al. [7]

(mapped as feedback on Task Level for Ott et al. [6]), but would add diversity in the format of feedback. Indeed, there is not a lot of help embedded in the exercise statement (i.e. not sensitive to the student's mistakes). However, since the prime objective of this thesis was to diversify the feedback types we did not develop this more.

For the second problem, the answer is in the task statement. Often they are capable of finding their mistake quickly and autonomously after reading the task again.

8.1.2 During the exercise

Another possible improvement that was thought of were identifying and correcting common misconceptions. We did not pursue this solution for several reasons. First, it was difficult to find a lot of information about different misconceptions. A useful site however, was *Progmiscon.org* [11], where several misconceptions are listed for programming languages such as Python or Java. The second problem was that we did not find examples of students having any of these misconceptions and making related mistakes in INGIInious exercises. Finally, a great obstacle was that identifying the symptoms of those misconceptions in student's code would prove itself to be quite difficult. If we wanted to address the misconceptions, it would have to be explained only in the theory syllabus and would not really add a lot of feedback on exercises.

In order to add more feedback of type How to Proceed, we thought about providing the student a few steps and asking him to put those steps in the correct order. We did not explore this further as this was a typical example of a Parson's problem, which another student was already developing in his master thesis.

A feedback of the feedback type Hint on Task Requirement that is often given, and the most frequent of its category, is "You did not name the method as expected.". This very general feedback is often caused by one of these two mistakes: either the student did a typo error, or they don't understand how a method or function is supposed to be structured. In the first case, the students often spots his mistake and corrects it. In the second case however, this feedback often does not help the students, as they don't know what they did wrong. This could be the ideal opportunity to provide some feedback of type Knowledge about Concepts, and explain the syntax of methods or functions. We did not develop this as we put the emphasis on Knowledge about How to proceed.

8.1.3 After the exercise

A level of feedback we did not mention a lot but that is very important is the self-regulatory level. This is harder to implement as exercise feedback. However, we thought about two ways self-reflection could be encouraged.

The first idea we had, was to show the student an ideal way of coding the solution to the specific problem once it has succeeded on his own. This way he can have a critic look on the style and the process behind another solution and his solution. Sometimes it would be better to just present a general way on how to proceed instead of real coding so that the students focuses more on the ideas behind the coding. Other times, actual code could be a better idea when we want to stress a good practice of coding style.

Another idea was to tease what the students would learn next and to put it in perspective with what they just saw. The goal is for them to take a step back and view their learning as a process with a bigger picture. This would also help them respond to the question "where to next" on the task level for the classification described in Ott et al. [6].

8.1.4 Other Exercises

The last idea we had was to create an exercise that would focus specifically on what the interpreter feedback meant. In the beginning students really struggle to understand how to interpret the errors generated by the interpreter. Helping them correctly identifying their errors would help them be more autonomous. Although it is strictly speaking not a feedback on the self-regulatory level, it does help them give themselves feedback and not rely on INGINious.

8.2 Conclusion

In the process of developing new feedback to diversify the types of feedback on the INGINious platform, various recurring patterns in exercise solutions have been identified. The aim was to be able to give the students a skeleton of how to approach the exercises and for them to be able to reuse one exercise-solving process onto another. To give concrete form to this feedback, six plans have been designed, each illustrated by a flowchart. Pseudo-codes have been developed for each data structure and each plan to make this approach more practical and show the correspondence with the actual code. Additionally, variants of these pseudo-codes replacing the for loop with a while loop have been presented. In this thesis, the

insertion of these diagrams in the theory syllabus has been presented, along with the color code used. Three types of support for the INGIInious exercises have been developed: scaffolding, hints based on plans and hints based on a previous exercise. Their inclusion on the INGIInious platform has been discussed.

The validation showed that although these diagrams could be useful, especially as a starting point for coding, there was still room for improvement in terms of their clarity and the links between them. It would also be good to specify and deepen the insertion in the course, whether in the theoretical syllabus or in the practical sessions, in order to support the concepts behind them.

Overall, the plans we developed appear to be useful and to have achieved our objectives of providing more feedback on a Process level. The help we provide students are enabling them to take a step back and to observe the similarities in process for solving exercises as well. The feedback provided also enables them to start the coding process of solving an exercise when feeling lost. The clarity of those diagrams and the way they are inserted into the course could be improved and the downside of the patterns is their limited use, both in regards to the exercises as to the overall programming journey of students. However, to a lesser extent, it could motivate students not to give up when they don't know how to start as well as deepen their understanding of coding early on in their discovery of this new skill.

Bibliography

- [1] W. Lewis Johnson. *Intention-based Diagnosis of Novice Programming Errors*. en. Google-Books-ID: dIPE6MQF3uEC. Morgan Kaufmann, 1986. ISBN: 978-0-934613-19-4.
- [2] D. Royce Sadler. “Formative assessment and the design of instructional systems”. en. In: *Instructional Science* 18.2 (June 1989), pp. 119–144. ISSN: 1573-1952. DOI: 10.1007/BF00117714. URL: <https://doi.org/10.1007/BF00117714> (visited on 05/21/2023).
- [3] Allen B. Downey, Jeffrey Elkner, and Chris Meyers. *How to think like a computer scientist: learning with Python*. en. 1. ed. Wellesley, Mass: Green Tea Press, 2002. ISBN: 978-0-9716775-0-0.
- [4] John Hattie and Helen Timperley. “The Power of Feedback”. en. In: *Review of Educational Research* 77.1 (Mar. 2007), pp. 81–112. ISSN: 0034-6543, 1935-1046. DOI: 10.3102/003465430298487. URL: <http://journals.sagepub.com/doi/10.3102/003465430298487> (visited on 05/30/2023).
- [5] Susanne Narciss. “Feedback Strategies for Interactive Learning Tasks”. In: Jan. 2008, pp. 125–144.
- [6] Claudia Ott, Anthony Robins, and Kerry Shephard. “Translating Principles of Effective Feedback for Students into the CS1 Context”. en. In: *ACM Transactions on Computing Education* 16.1 (Feb. 2016), pp. 1–27. ISSN: 1946-6226. DOI: 10.1145/2737596. URL: <https://dl.acm.org/doi/10.1145/2737596> (visited on 05/04/2023).
- [7] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. “A Systematic Literature Review of Automated Feedback Generation for Programming Exercises”. In: *ACM Transactions on Computing Education* 19.1 (Sept. 2018), 3:1–3:43. DOI: 10.1145/3231711. URL: <https://doi.org/10.1145/3231711> (visited on 03/30/2023).

- [8] Olivier Goletti, Kim Mens, and Felienne Hermans. “An Analysis of Tutors’ Adoption of Explicit Instructional Strategies in an Introductory Programming Course”. In: *Koli Calling ’22: 22nd Koli Calling International Conference on Computing Education Research*. Koli 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1–12. ISBN: 978-1-4503-9616-5. DOI: 10.1145/3564721.3565951. (Visited on 11/18/2022).
- [9] *INGInious, une aide précieuse à l’apprentissage de l’informatique*. fr. URL: <https://uclouvain.be/fr/facultes/epl/inginius-une-aide-precieuse-a-l-apprentissage-de-l-informatique.html> (visited on 05/21/2023).
- [10] *Interactive syllabus: Index*. URL: <https://syllabus-interactif.info.ucl.ac.be/index/info1-theory> (visited on 05/30/2023).
- [11] *Programming Misconceptions*. en. URL: <https://progmiscon.org/misconceptions/Python/> (visited on 05/21/2023).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl