Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sigmund William Cook
Ole-Christian Bjerkeset

Common mistakes made by novice programmers

A comparison between Java and Python

Master's thesis in Informatics
Supervisor: Guttorm Sindre
June 2023

NTNU
Norwegian University of
Science and Technology

Sigmund William Cook
Ole-Christian Bjerkeset

Common mistakes made by novice programmers

A comparison between Java and Python

Master's thesis in Informatics
Supervisor: Guttorm Sindre
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Programming remains a challenging subject to learn and teach, with students often encountering a variety of errors as they navigate through the intricacies of languages such as Java and Python. Our study provides a comprehensive analysis of these common errors made by novice programmers in both languages, and explores the correlation between them. Additionally, we compare the error patterns between different exam contexts, namely home and school exams.

The central research questions we aim to answer are:

1. What are the common errors made by novice programmers in Java?
2. What are the common errors made by novice programmers in Python?
3. Is there a correlation between the common errors made by novice programmers in Java and Python?
4. Are there differences in the errors made during home exams compared to school exams?

Our findings indicate that while some errors are unique to a specific language, many others are prevalent across both languages, suggesting a shared set of challenges for novice programmers. We have also identified subtle differences in error occurrence between home and school exams, underscoring the potential influence of the learning environment on programming difficulties.

These results hold significant implications for educators, helping them design targeted interventions that can address the common areas of difficulty more effectively. Further, the insights on the impact of the learning environment on error patterns can guide the adaptation of teaching strategies according to the context.

The detailed presentation of our findings not only contributes to the existing body of literature on computer science education, but also serves as a practical guide for educators seeking empirical evidence to enhance their teaching strategies. The potential correlations and differences identified through our research can form the basis for further exploration and hypothesis formation in future studies.

Acknowledgements

We want to thank our supervisor Guttorm Sindre for his patient guidance over the past year. Thank you for all the nice meetings and great feedback on both the research and writing of the thesis.

Contents

Abstract . . . . . **i**
Acknowledgements . . . . . **ii**
Contents . . . . . **iii**
Figures . . . . . **vi**
Tables . . . . . **vii**
1 Introduction **1**
1.1 Objectives 1
1.1.1 Motivation 1
1.1.2 Research Question 2
1.2 The Scope of The Study 2
1.3 Thesis Overview 2
2 Literature review **4**
2.1 Java 4
2.1.1 Automatic Tools 4
2.1.2 Surveys and Interview 8
2.1.3 Combination of Automatic Tools and Surveys/Interviews . . 11
2.2 Python 16
2.2.1 Automatic Tools 16
2.2.2 Surveys and Interviews 20
2.3 Other Programming Languages 21
2.4 Summary and Findings 22
2.4.1 Research Questions 23
2.5 Contribution 24
3 Methodology **25**
3.1 Research Design 25
3.1.1 Python Script and Unittest Customization 26
3.1.2 Generalizability 26
3.1.3 Depth and Breadth of Data 26
3.1.4 Efficiency and Resource Allocation 27
3.1.5 Validity and Credibility 27
3.2 Literature Review 27
3.2.1 Finding Papers 28
3.3 Data Collection and Analysis 29
3.3.1 Comparing Java Results 29

3.3.2 Manual Analysis of Python 31
3.3.3 Data Visualization 33
3.3.4 Categorising Data 34
3.3.5 Data Analysis 36
3.3.6 Qualitative Analysis 37
3.3.7 Python Script Development 37
3.3.8 Ethical Considerations 38
3.3.9 Conclusion 38
4 Results and Findings 39
4.1 Summarising Errors 39
4.1.1 Interpreting Results 40
4.1.2 Interpreting Results 42
4.1.3 Interpreting Results 44
4.1.4 Comparative Analysis 45
4.2 Proportions 47
4.2.1 Error proportions 47
4.2.2 Manual Analysis of Python Code Results 51
4.2.3 Java Results 52
5 Discussion 54
5.1 General Overview 54
5.1.1 Analysis of Autumn 2021 Home Exam 54
5.1.2 AssertionErrors in Exam Patterns 54
5.1.3 Analysis of Autumn 2022 55
5.1.4 Concluding Observations 55
5.2 Error Categories Across Exams 55
5.2.1 Autumn 2021 Home Exam 55
5.2.2 Autumn 2022 Morning Exam 55
5.2.3 Autumn 2022 Evening Exam 56
5.2.4 Interpreting the Data of 2022 Evening and Morning Exams . 56
5.3 Error Distribution Across Tasks 57
5.3.1 Task Challenges 57
5.3.2 Task Challenges 58
5.3.3 Task Challenges 60
5.4 Identifying Common Error Types 60
5.4.1 Errors in Autumn 2021 Exam (Eksamen H2021) 60
5.4.2 Errors in Autumn 2022 61
5.4.3 Errors in Autumn 2022 Evening Exam (Eksamen S2022 Kveld) 61
5.5 Commonalities and Differences in Errors Across Home and School Exams 61
5.5.1 Common Errors 61
5.5.2 Differences in Errors 62
5.5.3 Implications 63
5.6 Summary 64
6 Conclusion 65

6.1 General Findings 65
6.2 Limitations 66
6.3 Future Research 67
6.3.1 Potential Research Directions 67
6.4 Connection with Research Questions 68
6.5 Revisiting the Introduction 69
Bibliography **70**
A Exam Tasks and Suggested Solutions **74**

Figures

3.1 Example of a test case 30
3.2 Example of code 30

4.1 Home Exam 2021 48
4.2 School Exam Morning 2022 48
4.3 School Exam Evening 2022 49

A.1 Exam Tasks and Suggested Solutions Home Exam 2021 75
A.2 Exam Tasks and Suggested Solutions Evening School Exam 2022 . . 76
A.3 Exam Tasks and Suggested Solutions Morning School Exam 2022 . 77

Tables

2.1 Top 10 programming mistakes, Tabanao *et. al.*[6] 5
2.2 Top 8 programming mistakes, Mow *et. al.*[8] 6
2.3 Top 10 programming mistakes, McCall *et. al.* 2014 + 2019[11][10] 7
2.4 List of 20 errors defined by Hristova, *et. al.*[12] 9
2.5 Common themes throughout the interviews, Kaczmarczyk *et. al.*[15] 10
2.6 Top 10 programming mistakes, Caceffo *et. al.*[16] 11
2.7 Top 10 programming mistakes, Jackson *et. al.*[18] 12
2.8 Frequency of mistakes made by students in 2014, 2015 and 2017, Brown *et. al.*[19][20][21] 14
2.9 Frequency of errors and bad habits, Bulmer *et. al.*[22] 15
2.10 Frequency of mistakes between different achievement levels, Jegede *et. al.*[23] 16
2.11 Summary of findings from qualitative analysis, Veerasamy *et. al.*[24] 17
2.12 Top 10 programming mistakes Smith *et. al.*[25] 18
2.13 Top five programming mistakes Kohn[26] 19
2.14 Top 10 programming mistakes Jeffries *et. al.*[27] 20

3.1 Sample size and selection of exam submissions for analysis 29

4.1 Counts of Unique Errors 40
4.2 Counts of Unique Errors 42
4.3 Counts of Unique Errors 44
4.4 Caption 46
4.5 Sum of Error Categories 47
4.6 Sum of Error Categories 47
4.7 Error Categories Proportions in Exams 47
4.8 Sum of errors and percentage of students that made an error for each task home exam 2021 50
4.9 Errors and error percentages for each task morning exam 2022 . . . 50
4.10 Errors and error percentages for each task evening exam 2022 . . . 50
4.11 Results of manual analysis of Eksamen H2021 51
4.12 Results of manual analysis of Eksamen S2022 Morgen 51
4.13 Results of manual analysis of Eksamen S2022 Kveld 51
4.14 Frequency of errors in Java submissions from the NTNU study . . . 53

Chapter 1

Introduction

1.1 Objectives

1.1.1 Motivation

The motivation behind this research stems from the significant challenges faced by novice programmers when learning to code. Programming requires not only a solid grasp of syntax and language constructs but also the ability to think algorithmically and solve problems systematically. Novice programmers often encounter difficulties in translating their ideas into functional code, resulting in common errors and misconceptions.

Understanding the specific errors made by novice programmers in Java and Python is essential for several reasons. Firstly, it allows educators to design more effective instructional strategies that address the specific challenges faced by students. By targeting these common errors, instructors can provide targeted interventions, clarifications, and examples to help students overcome misconceptions and improve their programming skills.

Secondly, gaining insights into the common errors made in Java and Python can inform the development of educational materials, including textbooks, tutorials, and online resources. By identifying the most prevalent errors, content creators can focus on addressing these areas in a comprehensive and accessible manner, enhancing the learning experience for novice programmers.

Furthermore, understanding the differences and similarities in the errors made between Java and Python can provide valuable insights into the unique challenges associated with each programming language. This information can guide educators in choosing appropriate programming languages for introductory courses and tailoring their instructional approach to maximize student learning.

By examining both home and school exam errors, this research aims to shed light on any distinctions that may arise in these different contexts. Understanding these differences can help instructors adapt their teaching strategies and assessments to better align with the challenges students face during different evaluation scenarios.

Overall, this research project seeks to contribute to the field of programming

education by providing a comprehensive analysis of the common errors made by novice programmers in Java and Python. By addressing these errors and misconceptions, we aim to improve the teaching and learning of programming, ultimately leading to more competent and confident programmers in the future.

1.1.2 Research Question

Considering the complexities and intricacies involved in programming, especially for novice programmers, our study is guided by the following research questions:

- Q1: Which common errors are made by novice programmers in Java?
- Q2: Which common errors are made by novice programmers in Python?
- Q3: Are there any correlation between common errors made by novice programmers in Java and Python?
- Q4: Are there differences in the errors made during home exams and school exams?

1.2 The Scope of The Study

This study primarily focuses on identifying and analyzing the common errors made by novice programmers, particularly in Java and Python programming languages. The research is grounded in the context of the Norwegian University of Science and Technology (NTNU). Specifically, we have examined student submissions from both home and school exams in two introductory courses from NTNU. One course in object oriented programming using Java, and one using Python.

Our analysis involves categorization and comparison of errors across different conditions, highlighting the challenges faced by beginner students in their initial encounter with programming. However, it is worth noting that while our research attempts to provide a broad and comprehensive understanding, it is bounded by the sample data derived from these specific courses at NTNU. Therefore, the results may not encapsulate all possible programming errors made by novice programmers globally, or in other programming languages. However, our research should offer valuable insights into the struggles of first-time programming students at NTNU.

1.3 Thesis Overview

The structure of this thesis is as follows:

- Chapter 1 introduces the research team, presents the goals and the scope of the study, and provides an overview of the thesis.
- Chapter 2 offers a comprehensive review of the relevant literature and frames the context of this study within the broader field of computer science education research.

- Chapter 3 details the methodology adopted for this research, explaining the mixed-methods approach and the data collection and analysis procedures, as well as the methodology used when conducting the literature review.
- Chapter 4 presents the findings of the study, highlighting the common errors identified in Java and Python programming, as well as presenting some examples of common errors.
- Chapter 5 discusses the implications of the findings, considering their relevance for programming education, and explores potential avenues for future research.
- Chapter 6 concludes the thesis, summarizing the key points and reflecting on the overall research journey.

This layout aims to provide a logical flow of ideas and findings, offering readers a coherent and comprehensive understanding of the research.

Chapter 2

Literature review

There is a substantial amount of research focused on understanding misconceptions and errors made during programming, particularly among novice student programmers. These studies span a variety of programming languages, including Python and Java, and consider a diverse range of demographics and methodologies. However, there's a wide spectrum of combinations of languages, demographics, and methods that are yet to be fully explored. Delving into these different combinations could offer valuable insights to enrich the current body of research in this area.

2.1 Java

The study of common errors in Java programming reveals a considerable amount of diversity among different research papers. Predominantly, the data collection methods are bifurcated into two major categories: automatic tools and surveys/interviews, or a combination of both, presenting a natural division among various studies. This subsection includes some information derived from our prior work on this subject matter [1]. This variation in data collection approaches adds depth to the research, but also points to the need for uniform methodologies to ensure the comparability of findings across studies.

2.1.1 Automatic Tools

Automatic tools are highly efficient for identifying errors in code. Numerous studies have leveraged these tools to analyze common mistakes made in Java programming. The following are a selection of projects that closely align with our research focus.

Jadud attempted to identify the most common errors made by novice programmers, when he explored novice compilation behaviour in 2006. By focusing entirely on the "edit-compile" interactions students had with the Java compiler, he made observations and gathered data he could use for his research. He recruited 130 first year students in the University of Kent and made them work in BlueJ, a pedagogic programming environment intended to support the learning of Java from an objects-first perspective [2][3]. BlueJ allowed him to gather snapshots

of the students' code every time they compiled their program, along with output from the compiler and other useful metadata. The data was later used in different analyses, e. g. counting the occurrences of different errors and capturing the time between each compilation of errors [4]. He included a graph showcasing the most common errors encountered in his article, but unfortunately the quality of it was too low to be able to read. However, we can assume that it was mostly compilation and run-time errors, because all the data was gathered directly from the compiler.

Tabanao *et al.* conducted a study aimed at early identification of at-risk novice programmers during their initial stages of learning to code. To achieve this, the researchers collected compilation data and tracked errors made during five exercise sessions, utilizing a customized extension developed for BlueJ. Subsequently, the data was analyzed and compared with the students' midterm exam scores to explore potential correlations between the quantity and types of errors made and their performance on the exam. Additionally, the researchers examined the overall frequency of errors and compiled a comprehensive list of the top 10 most commonly encountered error types. There were a total of 52 different error types encountered and they based their categories on the compiler error message itself. The list of top 10 errors can be seen in table 2.1 below. Although the study yielded more questions than answers, the researchers did find a correlation between a low incidence of syntax errors and higher midterm exam scores [5]. Building upon this work, the same authors further attempted to predict at-risk students by developing a linear regression model. Unfortunately, the researchers were unable to achieve accurate predictions, but gained valuable insights regarding the compilation behaviour of the students [6]. Additionally, they used the same data in similar research where they tried to study the behaviour of novice programmers [7].

Table 2.1: Top 10 programming mistakes, Tabanao *et. al.*[6]

Mistake	**Frequency (%)**
cannot find symbol - variable	20
; expected	13
(or) or [or] or { or } expected	10
missing return statement	8
cannot find symbol - method	6
illegal start of expression	6
incompatible types	4
<identifier>expected	4
class, interface, or enum expected	3
else without if	2

Under the leadership of Ioana Tuugalei Chan Mow, the Computing department staff at the National University of Samoa conducted an extensive analysis of student errors in three distinct undergraduate Java programming courses. The study involved the meticulous logging of errors directly from the compilers and subsequent categorization

of these errors based on their types. The cumulative number of unique errors encountered ranged from 70 to 171 across the three courses. By meticulously tallying the frequency of each specific error, the researchers identified and summarized the top eight most prevalent mistakes observed throughout the three courses, as presented in table 2.2. To provide further insights, the errors were classified into three distinct categories: syntax, semantic, and logic errors. Interestingly, the study revealed that a significant majority of the errors, 94.1%, fell within the syntax category, while semantic and logical mistakes accounted for only 4.7% and 1.2%, respectively [8].

Table 2.2: Top 8 programming mistakes, Mow *et. al.*[8]

Mistake	**HCS181**	**HCS281**	**HCS286**	**Total**	**Frequency (%)**
Variable not found	101	18	41	160	49.8
Identifier expected	23	22	0	45	14.0
Class not found	12	3	1	16	5.0
Mismatched brackets/parentheses	11	6	0	17	5.3
Invalid method declaration	7	5	0	12	3.7
Illegal start of type	5	6	0	11	3.4
; expected	2	4	1	7	2.2
Method not found	4	1	0	5	1.6

McCall and Kölling conducted a comprehensive investigation of common programming mistakes, utilizing the Blackbox project as a valuable resource. The Blackbox project, initiated in 2013, aimed to facilitate the availability and reusability of research-friendly programming data. It operates as an ongoing data collection initiative, gathering anonymous data from global users through the BlueJ IDE. The collected data is made accessible to other researchers for their own studies [9].

The researchers conducted two distinct studies, with the latter building upon the findings of the former. In their pursuit of more precise categorization of student errors, McCall and Kölling opted for manual analysis rather than relying on automated tools and diagnostics for error classification. The data collection occurred in two phases, both utilizing the BlueJ IDE and encompassing Java code specifically written in BlueJ. During the first phase, data was gathered from 240 volunteer students enrolled in introductory Java courses at two different universities. This initial dataset was used to develop error categories before acquiring additional data through the Blackbox project, enabling a more extensive and diversified analysis. Throughout the category development phase, the researchers manually refined and structured the error categories, creating a hierarchical system that varied in specificity within the hierarchy. This entire process was data-driven and carried out manually. Subsequently, the second dataset underwent three distinct analyses: error frequency, number and frequency of diagnostic messages, and coverage levels of the most prevalent error categories. The results of these analyses are presented in table 2.3, showcasing the top 10 most frequent errors [10].

McCall and Kölling concluded that they made improvements on previous methodologies by analysing logical errors instead of diagnostic messages, claiming it to be a more reliable ranking of student error frequencies. They also pointed out a difference between their own results and studies conducted prior to theirs, giving rise to the hope that the production of automated diagnostic messages could be improved [11].

Five years later, McCall and Kölling expanded their research by collecting and analyzing a little over 1000 compilation events with errors, extracted from 199 user sessions. They developed a new hierarchical structure for error categorization, consisting of 11 primary error categories with numerous subcategories (a total of 90). This hierarchical structure was based on their earlier study conducted in 2014. A single researcher manually categorized the compilation events, forming the basis for subsequent analysis. The study examined error frequency, severity, and difficulty. The top 10 most frequent errors are presented in table 2.3. An intriguing finding from this study revealed that the top 10 logical errors accounted for 60% of all error occurrences. The researchers argue that by improving error diagnostics for these top 10 errors, students would benefit from enhanced feedback in 60% of error instances [10].

Table 2.3: Top 10 programming mistakes, McCall *et. al.* 2014 + 2019[11][10]

Mistake	**Freq. 2014 (%)**	**Freq. 2019 (%)**
Variable not declared	11.1	8.4
; missing	10.3	7.3
Variable name written incorrectly	8.4	7.4
Simple syntactical error	7.9	6.5
Method name written incorrectly	4.9	4.6
Missing parenthesis for constructor call	4.1	x
Unhandled exception	3.0	x
Class name written incorrectly	2.7	x
Method call: parameter type mismatch	2.4	x
Type mismatch in assignment	2.4	x
Semantic error	x	4.8
Variable: incorrect variable declaration	x	4.7
Variable: incorrect attempt to use variable	x	4.6
Method: incorrect method declaration	x	4.5
Class or type name written incorrectly	x	4.4

Comparing the tables of the top 10 most frequent errors from both research projects conducted by McCall and Kölling yields interesting insights. As depicted in the above table, the distribution of error frequencies exhibits a striking similarity between the two studies. This similarity could directly reflect the discrepancy in results or be influenced by factors such as slight alterations in the category definitions. Notably, the second project placed greater emphasis on manual analysis,

offering a valuable opportunity to compare the accuracy of automatic error detection by contrasting the outcomes of both papers.

The fact that the error distribution remained relatively unchanged between 2014 and 2019 could indicate the accuracy of automated error detection. However, it is essential to consider other contributing factors that may have influenced these findings. Furthermore, it is intriguing to note that the five most common errors identified in the 2014 research experienced a significant decrease in frequency in the 2019 study. This decline can be attributed to various factors, including improved educators and the overall quality of education, students' increased knowledge, and advancements in technology. For instance, integrated development environments (IDEs) have undergone notable enhancements over the years, incorporating features such as syntax error warnings and auto-completion of variable names. These improvements significantly enhance the coding experience, potentially contributing to the reduction of common errors in the later study.

2.1.2 Surveys and Interview

Surveys and interviews serve as effective methods for gathering both quantitative and qualitative data, making them valuable for identifying common programming errors. When utilizing these approaches, insights can be obtained from students and experienced educators, providing firsthand experiences that contribute to the identification and understanding of prevalent programming mistakes. The following research papers showcase the use of surveys and/or interviews as their primary research methods:

Hristova *et al.* conducted a study where they collected data from various professors and professionals to identify the most common Java mistakes made by novice students. Based on the results, they compiled a list comprising 20 distinct errors, which were subsequently categorized into three main categories: syntax, semantic, and logic errors. Additionally, the researchers developed an error detection system. Unfortunately, as they did not collect data directly from students, they did not verify the accuracy of the identified errors or the functionality of the implemented tool. Nevertheless, the list of mistakes and the developed tool remain valuable resources that could be utilized in subsequent research endeavors. Table 2.4 presents the list of identified mistakes [12].

Table 2.4: List of 20 errors defined by Hristova, *et. al.*[12]

ID	**Mistake**	**Type**
A	= vs. ==	Syntax
B	== vs. .equals	Syntax
C	Mismatching, miscounting and/or misuse of {}, (), [], " ", and ' '	Syntax
D	&& vs. & and || vs. |	Syntax
E	Incorrect semi-colon after *and if*-selection structure before the *if*-statement or after the *for* or *while* repetition structure before the respective *for* or *while* loop	Syntax
F	Wrong separators in *for* loops (using commas instead of semi-colons)	Syntax
G	An *if* followed by a bracket instead of by a parenthesis	Syntax
H	Using keywords as method names or variable names	Syntax
I	Invoking parentheses after method call	Syntax
J	Forgetting parentheses after method call	Syntax
K	Incorrect semicolon at the end of a method header	Syntax
L	Leaving a space after a period when calling a specific method	Syntax
M	>= and =<	Syntax
N	Invoking class method or object	Semantic
O	Improper casting	Logic
P	Invoking a non-void method in a statement that requires a return value	Logic
Q	Flow reaches end of non-void method	Logic
R	Methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation	Logic
S	Incompatibility between the declared return type of a method and in its invocation	Logic
T	Class declared abstract because of missing function	Logic

A notable study by Garner *et al.* in 2005 took a slightly different approach, focusing on identifying common problems encountered by novice programmers. The researchers developed a predefined list of 27 problems, which served as a basis for data collection. Surveys were administered to teaching assistants following student programming lab sessions. The teaching assistants recorded the specific problems they had to address while assisting students and mapped them to the predefined list of 27 problems. The study revealed an interesting finding: a significant dominance of the problem category termed "Basic mechanics." This category encompassed simple syntactical errors such as missing semicolons, highlighting the prevalence of such mistakes among novice programmers [13].

Another relevant study conducted by Kaczmarczyk *et al.* employed formal interviews with students to identify common misconceptions among novice programmers. The interviews took place at the University of California, San Diego, and involved 11 undergraduate students enrolled in CS1 courses during the spring of 2009. From a concept inventory comprising 32 entries developed by Delphi experts in 2008 [14]. Kaczmarczyk *et al.* selected 10 concepts as the focus of their interviews. The primary objective of the interviews was to identify misconceptions related to these 10 concepts, while the secondary objective was to validate the conclusions of the Delphi experts. Each student was presented with a subset of 18 problems covering the 10 concepts during the one-hour interview session, which encompassed questions on all the topics. The analysis of the interview results revealed four prominent themes, as illustrated in table 2.5.

Theme 1 and 4 revolve around language-specific misunderstandings and cover general misconceptions. Theme 2 encompasses various misconceptions related to an inadequate understanding of how while loops function, which is not entirely language-dependent. Theme 3 represents a fundamental lack of understanding of key aspects of Object-Oriented Programming. Within these themes, three significant misconceptions were identified and described in detail: "Semantics to semantics," "Primitive no default," and "Uninstantiated memory allocation." The researchers concluded that the gathered data provided valuable insights for instructors in real-time. Additionally, the results were merged with additional data to be collected and used in the development and validation of a CS1 curriculum for Programming Fundamentals [15].

Table 2.5: Common themes throughout the interviews, Kaczmarczyk *et. al.*[15]

T1	Students misunderstand the relationship between language elements and underlying memory usage.
T2	Students misunderstand the process of while loop operation.
T3	Students lack a basic understanding of the Object concept.
T4	Students can not trace code linearly.

Caceffo *et al.* pursued a distinct methodology to identify misconceptions in Java by leveraging already established misconceptions in C and Python. Their approach involved converting these existing misconceptions, where feasible, to the context of Java. Following the conversion process, an open response test comprising open-ended questions was administered to validate the proposed misconceptions. A total of 27 misconceptions were identified and considered in the open response test. Out of the 111 students who participated, 22 of these misconceptions were observed in their responses. Additionally, the test revealed six new potential misconceptions, expanding the pool of possible misunderstandings to 28. The occurrence frequency of each programming mistake varied widely, ranging from one to 21 instances among the total surveyed students. The researchers recorded the frequency of each programming mistake, with the top 10 most frequent mistakes presented in table 2.6.

The paper also addressed the language independence of misconceptions identified through Concept Inventories. Notably, they found that a higher percentage of misconceptions (32%) were exclusively mapped from C to Java, as compared to Python (10%). This observation highlights the nuanced nature of misconceptions and the variations that arise when transitioning between programming languages [16].

Table 2.6: Top 10 programming mistakes, Caceffo *et. al.*[16]

Mistake	**Frequency (%)**
No return value in a function that returns something	18.9
Attempt to access parameter from outside scope	10.8
Class attribute invoked without being imported or with no class specified	9.0
Global variables assumed inaccessible from within function	5.4
Arithmetic expression instead of boolean expression	5.4
No parameters used when calling a method	4.5
Attempt to access local variables, except parameters, from outside scope	4.5
Wrong order in logical or arithmetic operations	4.5
Nested if-statements instead of boolean expression	3.6
Incorrect order of function parameters	2.7

2.1.3 Combination of Automatic Tools and Surveys/Interviews

Several projects have employed a combination of both automatic tools and surveys/interviews to gather qualitative and quantitative data in their efforts to identify common mistakes in Java. The integration of these methods provides researchers with comprehensive insights. The following papers demonstrate the utilization of these combined approaches:

A study conducted on freshmen at the United States Military Academy focused on identifying the most common Java errors among novice programmers. The researchers implemented a real-time automated error detection system called "Gauntlet", which logged all errors to a central database. Gauntlet had been developed approximately two years prior to the study [17]. Over the course of one semester, the system collected a total of 559,419 errors. Among these errors, the top 10 accounted for 51.8% (290,134) of the total. The use of an automated system enabled the researchers to precisely track each specific type of Java error, resulting in highly accurate results. The top three errors, in descending order, were "cannot resolve symbol," ";expected," and "illegal start of expression," with the first error being significantly more frequent. The remaining top 10 errors are detailed in table 2.7. Furthermore, the researchers conducted interviews with faculty members to compare the errors identified by the automated system with those recognized by the faculty. An interesting finding emerged: a discrepancy existed between the errors identified by the automated system and those recognized by the faculty. This discrepancy suggests that automated systems may have value in identifying

errors and misconceptions. Alternatively, it could be attributed to the faculty's focus on significant logical errors made by students, while potentially paying less attention to simple syntactical errors [18].

Table 2.7: Top 10 programming mistakes, Jackson *et. al.*[18]

Mistake	**Frequency**	**Faculty identified**
cannot resolve symbol	81655	Yes
; expected	47362	Yes
illegal start of expression	32107	No
class or interface expected	25650	No
<identifier expected>	25224	Yes
) expected	21412	Yes
incompatible types	15854	No
int	14185	No
not a statement	13878	No
} expected	12808	Yes

By combining methodologies previously mentioned, data set of student-written code, automated error detection and surveys, Brown and Altadmri made three separate attempts to identify the most common novice Java programming mistakes. The first one, being in 2015, they decided to use previous work done by Hristova *et. al.*[12] as a base for their research. They took the 20 student mistakes defined by Hristova and removed two of them; first one being "leaving a space after a period when calling a method"(L), because it was not a programming mistake in their eyes, and the second one being "improper casting"(O) because it was not described clearly enough. After defining their list of errors they created a survey asking educators to rate each mistake on a scale of infrequent to frequent, by making a mark along a visual analogue scale. The scales were then measured to the nearest $\frac{1}{100}$ of their length and recorded as a number between zero and 100. The questionnaire was filled out by a total of 76 educators, forming the sample set for the analysis of the educators' beliefs. Student data was taken from the Blackbox data set previously mentioned [9]. They used three different methods to detect the 18 different student mistakes; the compiler error message, a post-lexing analysis and a customised permissive parser. They then tracked the source file over time, checking each compilation for one or more of the 18 mistakes and counting their occurrences. The results from the survey showed that the educators form a very weak consensus about which errors are most frequently made by students. The results from the student data analysis can be seen in table 2.8 below. They also concluded that educators are not very accurate compared to the large data set of student mistakes and that an educator's level of experience had no effect on how closely the educator's frequency rankings agreed with those from the dataset. They continued the argument by claiming that these errors could be contextual,

meaning each educator are correct about their own students mistakes [19].

During the second project they only used student data in their research, once again gathered from the Blackbox dataset. They used the exact same mistake categories previously defined as well so the setup was fairly similar, only this time they had more data. One key difference was the fact that they included an estimated time to fix based on the compilation data. The resulting frequencies of mistakes can be seen in table 2.8 below, we have not included the median time to fix for each mistake due to lack of relevancy to our own research. They concluded that we should be careful not to over-interpret the results as it is greatly affected by which mistakes they included in the study, but their results suggest that semantic errors are a more serious challenge than syntax errors [20].

For the third project they went back to the methodology used in 2014; a combination of student-written code, automated error detection and surveys. By using the same set of 18 pre-defined errors, they collected student source files from the Blackbox project, this time with even more compilation data. They recorded the frequency of errors from the automated tools which can be seen in table 2.8 below [21]. They used the exact same educator survey data they collected for their research paper in 2014, meaning that the results and conclusions surrounding the data from educators are exactly the same as 2014.

It is interesting to see that the top three most frequent mistakes remain the same over each research project. The frequency distribution in general are very similar over all the 18 categories as well. This could potentially be an indication that the education system have remained unchanged between the year 2014 and 2017. However, because every project used the Blackbox dataset there are most likely a vast amount of duplicate data, only difference being additional data being added in between years. This would also be an explanation for why the results correlate so much over the years.

It is also unfortunate that they did not include a percentage of the mistakes in the compilation data they collected, and without knowing whether they looked for mistakes in all of the compilations recorded, the successful compilations, or the unsuccessful ones, it is impossible to calculate. However, the total number of compilation events each data set consisted of was 14,235,239 in 2014, 37,158,094 in 2015 and around 100 million, assuming they used the entire Blackbox dataset, which gives us some indication of the normality for each mistake. Nevertheless, the research they did is very interesting and could potentially have value when attempting to optimize course development and improving teaching practices.

Table 2.8: Frequency of mistakes made by students in 2014, 2015 and 2017, Brown *et. al.*[19][20][21]

Mistake	**Type**	**freq. 2014**	**freq. 2015**	**freq. 2017**
C - mismatched parentheses	Syntax	404560	793232	1861627
I - calling method with wrong types	Type	165832	464075	1034788
O - missing return statement	Semantic	137230	342891	817140
N - discarding method return	Semantic	86107	121663	274963
A - confusing =with ==	Syntax	68254	173938	405748
B - using == to compare strings	Semantic	45012	121172	274387
M - invoking instance method as static	Semantic	30754	86625	202017
R - missing methods when implementing interface	Semantic	24846	79462	186643
P - including types in actual method arguments	Syntax	21694	52862	117295
E - spurious semi-colon after if, for, while	Syntax	20264	49375	108717
K - spurious semi-colon after method header	Syntax	16156	38001	86606
Q - type mismatch assigning method result	Type	14371	16996	32435
D - confusing &with &&	Syntax	11212	29605	61965
J - forgetting parentheses when calling methods	Syntax	8332	18955	43165
L - less-than/greater-than operators wrong	Syntax	1916	4214	9381
F - wrong separator in for	Syntax	1171	2719	6424
H - keyword as variable or method name	Syntax	415	1097	2568
G - wrong brackets in if	Syntax	63	118	284

Bulmer *et. al.* designed and built their own software to visualize code patterns amongst novice programmers in Java. The code visualizer was designed to animate the programming dynamically and summarize error metrics simultaneously. Their reasoning was because they did not want to work with submitted code, as it has usually gone through several iterations of changes before final submission. As such, they felt that it limited their ability to analyze the kinds of errors students come across throughout the coding process. As a start, the visualizer was designed to detect a small number of errors and bad habits. The list of said errors and habits was chosen based on a literature review and pas teaching experiences, and it can be seen below:

- **Unclosed Scanners:** Creating a Scanner object but never calling its close method
- **Brackets and Quotes Miscounts:** The overall number of open brackets or open single/double quotes being different from the number of close brackets or quotes
- **Brackets and Quotes Mismatches:** Closing a bracket or a quote with a different type of bracket or quote (e.g., “, [)”
- **Misplaced Semi-colons:** Placing a semi-colon after a condition in a conditional statement or after a loop declaration
- **Comparison vs. Assignment:** Confusing the equality and assignment operators
- **Misaligned Whitespaces:** Failing to indent consistently within a block of code

They gathered data from 77 students in a CS2-course, all of whom recently completed a CS1 course. Over the month of January 2018 a system was available online for the students, which included a series of exercises and a short confidence survey covering core concepts taught in CS1 courses. The confidence survey asked the students to rate their confidence level on 11 topics, while the exercises consisted of 10 programming questions from the same list of topics. In addition to the errors and habits mentioned above they also tracked the frequency of individual errors. confidence level of students, amount of code written and number of mistakes per user. The resulting frequency of errors and bad habits can be seen in table 2.9 below. Their analysis suggested that some of the bad habits result from students' overreliance on IDE features. Additionaly, they found that more confident students wrote more code, with fewer errors and bad habits. However, only 13 out of the 77 students completed all 10 exercises, while the rest of them only completed half, which could potentially affect the results of the study [22].

Table 2.9: Frequency of errors and bad habits, Bulmer *et. al.*[22]

Mistake/habit	**Frequency**
Unclosed scanners	293
Brackets and Quotes Miscounts	205
Comparison vs. Assignment	81
Brackets and Quotes Mismatches	44
Misaligned Whitespace	40
Misplaced Semi-Colons	11

Jegede *et al.* decided to use a different approach than the ones previously mentioned. Instead of only identifying mistakes made by novice Java programmers in general, they wanted to compare the mistakes made between; low, average and high achieving novice students. They only had 5 pre-defined groups of mistakes, which is really small compared to other studies. Results from the research showed that the top three most common mistakes altogether were "Missing Symbol", "Mismatched Symbol", and "Excessive Symbol" respectively. They also created multiple tables

with interesting statistics, one of which compared the frequency of mistakes made between different achievement levels, this can be seen in table 2.10 below. They argued that the error types were the same across all achievement levels. However, the low achieving students had a harder time writing bug free code in other object concepts. In addition to identifying the most common errors they also wanted to elicit information on the reasons leading to the errors. This was done by conducting an interview where they selected 12 out of the 124 students that took part in the research. However, there were no significant results or correlations found from the interviews [23].

We find it interesting that the high achieving student made a significant higher number of mistakes, compared to the average- and low-achieving students. However, there were a total of 65 students in the high achieving cohort, which is quite high compared to the total of 33 and 26 in the medium- and low-achieving cohorts, giving us a natural explanation. Instead of showing us a table of the total distribution of errors it would be interesting to see individual statistics for each cohort. Additionally, it would also be interesting to see this study being conducted on a different demographic for comparison.

Table 2.10: Frequency of mistakes between different achievement levels, Jegede *et. al.*[23]

	Invalid symbol	**Mismatched symbol**	**Missing symbol**	**Inappropriate name**	**Excessive symbol**	**Total**
Low achieving	8.7%	26%	40.2%	14.1%	11%	21.2%
Average achieving	11.1%	26.3%	28.1%	8.2%	26.3%	28.6%
High Achieving	13.7%	18.3%	32%	15.3%	20.7%	50.2%

2.2 Python

Similar to papers written on mistakes in Java, a natural division between different methods for gathering data can be found in papers on Python as well. Automatic tools and surveys/interviews are still the main methods, while a paper using a combination of both was not found during the literature review.

2.2.1 Automatic Tools

In a study conducted by Veerasamy *et. al.* in 2014, they gathered final e-exam answers from novice students taking an introductory programming course. There were 69 students enrolled in the course, however only 39 attended the final e-exam. The final exam contained two multiple choice questions, two code tracing questions and six code writing questions using python as the programming language.

After collecting all the exam data using VILLE collaborative tutorial software tool, the researchers performed both a qualitative and quantitative analysis on said data. In order to identify and categorise different types of misconceptions and errors, they derived 10 topics from a 32 item long concept inventory developed by Goldman *et. al.* using the Delphi method [14]. They later followed a pre-defined analysis protocol for the qualitative analysis, where they among other things classified the findings from all the exam answers based on the concept inventory of 10 topics. Additionally, they performed a quantitative analysis by counting the number of mistakes that were identified in the qualitative analysis using statistical methods. However, instead of using the concept inventory categories they decided to use three different types instead; knowledge-, skill-based- and rule-based error. Their results from the quantitative analysis showed that 69.2% made knowledge-based errors, 5.1% made skill-based errors and a few students made rule-based errors due to "assumption-based confusion" around the use of library functions in coding their answers. A summary of the findings from the qualitative analysis can be found in table 2.11 below [24]. Although Veerasamy *et. al.* used automatic tools in order to aid their analytical process they also did a lot of manual labour, especially during the qualitative analysis.

Table 2.11: Summary of findings from qualitative analysis, Veerasamy *et. al.*[24]

Key topics	**Misconception**
Syntax and semantics	Student misunderstood the meaning of inbuilt function and its application
Parameter scope	Student confused about the use of return statement and data type of the parameter passing
Writing expressions for conditionals	Student misunderstood the process of control flow statements, especially nested if
Tracing loop execution	The student misunderstand the process of for loop operation
Defining and referencing list elements	Student was not clear with index position and referencing list elements

A quantitative study on the mistakes of novice programmers were conducted by Smith *et. al.* in 2019. The collected a large corpus of student implementations of eight programming problems. These problems were collected from three massive open online courses (MOOCs) all specializing in introductory Python programming. All of the students developed their code in CodeSkulptor, which is a browser based IDE with the ability to save their progress in a cloud storage during code development. They collected in total over 330 000 implementations over 95 000 development chains, through an exhaustive data collection approach. The implementations were then evaluated through auto-generated test suites, that according to them, have been shown to provide better coverage than carefully hand-crafted tests. The total amount of errors from all implementations were 196 315 or 58.3%. It

should be noted that these implementations were gathered every time a student saved their progress instead of looking at the finished product, possibly making the code more error-prone. The errors collected were broken down into 16 different types of run-time errors and 3 other error categories for more details. The 10 most common mistakes of said error categories can be seen in table 2.12 below. They also provided insight into the duration and evolution of errors, causing clear patterns to emerge, pointing to areas that merit additional attention by instructors and researchers alike. They concluded that future studies should aim to tease out the underlying reasons behind these patterns to better inform the decisions instructors make as they help students overcome these mistakes [25].

Table 2.12: Top 10 programming mistakes Smith *et. al.*[25]

Mistake	**Frequency (%)**
SyntaxError	11.94
NameError	10.04
TypeError	8.92
IndexError	4.49
KeyError	4.46
ImportError	2.62
Timout	2.05
AttributeError	1.92
RuntimeError	1.31
UnboundLocalError	1.07

It is also worth to mention that they had a category called "Incorrect Result" with a frequency of 50.29%, but because these implementations were gathered from unfinished code we find a high frequency to be expected, which is why we decided to leave it out of the table.

As a precursor to providing improved error messages in Python, Kohn collected and analyzed Python programs written by high school students taking introductory programming courses. By making their Python editor public, students were able to send their error reports and copies of the corresponding programs anonymously to the researchers. They collected a total of 6981 raw error messages of which 4091 were relevant to the study. During the analysis, the researchers characterized and counted different mistakes and additionally which mistakes the students were able to fix. They used this information among other things, as an indicator of how effective the error messages were and to discover the most common mistakes amongst the students. A list of the top five most common mistakes can be seen in table 2.13 They concluded that a considerable part of the errors were due to minor mistakes, such as misspellings which are easy to fix. The nature of other errors or the students' responses however, cannot always be reliably determined without knowing the goals and plans behind the program. Moreover, the compiler is not even able to detect all syntax errors or correctly classify them. The reason

why enhancing CEMs (Compiler error message) so often appears to be ineffective might thus be due to many errors being easy to fix even without further help, while other error are not actually captured by the CEM at all. Additionally, if students try to fix the CEM rather than the error itself, the recurrence of CEMs, say, might not be a good indicator for a student's progress [26].

Table 2.13: Top five programming mistakes Kohn[26]

Mistake	**Frequency**	**Fixed**
Name Error: Cannot Find Name	1462	1252
Wrong/Inconsistent Indentation	608	485
Type Error	349	287
Missing Comma or Operator	249	197
Missing/Mismatched brackets	240	223
Total	4091	3428

Although this study mostly focused on the compiler error messages and its effectiveness we still get some insight into which programming mistakes are most common amongst novice students, making this study relevant to our work as well.

Another study conducted on online courses was done in 2022 by Jeffries *et. al.*. The data used in the analysis were collected from the NCSS challenge, an online programming course run for students at participating schools. The languague used during the course is python 3 with several different streams avaiable targeting different levels of prior experience. The analysis used data from the stream for beginner programmers, where most of the students were from high-school and some from primary school. There were 1,281,068 terminal runs recorded of which 328,476 (25.6%) resulted in errors reported by the python 3 interpreter and 3309 distinct messages in total. In order to understand the prevalence of the errors being made the distinct error messages were manually inspected to identify similar messages, grouping them together and reducing the set to 113 distinct messages used in the analysis. They analysed the frequency of each error message resulting in a table of top 10 errors made seen in table 2.14 below [27].

Table 2.14: Top 10 programming mistakes Jeffries *et. al.*[27]

Mistake	**Frequency**
SyntaxError: invalid syntax	40771
NameError: name — is not defined	14255
IndentationError: expected an indented block	11997
IndentationError: unexpected indent	8651
SyntaxError: EOL while scanning string literal	7529
TypeError: unsupported operand type(s) for —; '—' and '—'	7193
SyntaxError: unexpected EOF while parsin	5369
IndentationError: unindent does not match any outer indentation level	3377
TypeError: — takes — positional arguments but — given	2054
SyntaxError: Missing parentheses in call to 'print'	1505

We find it interesting that these studies were conducted on MOOCs which often represents vast demographic differences. The students they collected data from hailed from 180 different countries and ranged from the ages 16 to 94, with an average of 39. This diversity could be argued as a strength, however we could also argue towards the potential threat to internal validity of the results, as the technological skill-set of the online students might be lesser than those of a college student. These studies were also conducted on CS0 courses, whereas other similar studies are usually conducted on CS1 or higher level courses.

2.2.2 Surveys and Interviews

Johnson, along with a few other professors from the University of Glasgow, conducted a study where they documented their observations of level 1 university students over several years. By teaching Python to a large cohort of first year students over many years, the researchers were enabled to identify common misconceptions amongst the novice programmers. By documenting their own observations and supplying this with a survey designed to test the robustness of the mental models of students that had been taught Python, they built evidence that their anecdotal findings were more widespread and attempted to probe the nature of said misconceptions. Based on the results from their first survey, carried out in 2018, they refined their original hypothesis, and shifted their focus from obscure parts of the Python language. In consequence, they developed and carried out a new survey in 2019. It was the findings from their second survey they discussed in the paper leading up to their conclusion. The survey consisted of questions posing a Python fragment, asking for the output or final value of the variables from the code snippets. Additionally, each question asked for the degree of confidence in all of each respondent's answer

and for eventual additional comments or observations they may have had. The survey attracted 42 responses, largely from students participating in a first year Python programming course. Due to space constraints the researchers did not cover all of their discussions in the paper, making it unclear which misconceptions that were scrutinized during their research.

Broadly, they found that the confidence of the respondents matched the correctness of their answers, concluding that this was a promising results; speaking on the danger for program quality if a programmer is confident in their incorrect knowledge. They also concluded that many students lack a clear understanding of many of Python's core language constructs. This lead up to their hypothesis that while teaching Python alone is insufficient, teaching it alongside visual languages that share a common notional machine improves students' understanding [28].

2.3 Other Programming Languages

In addition to research conducted on students programming in Java, there are also several research papers that look into students programming in other languages. One example is a study by Ettles et al. This study focuses on common logic errors made by novice programmers programming in C. The study analyzed 15,000 code fragments created by novice programming students that contained logic errors, and classified the errors into algorithmic errors, misinterpretations of the problem, and fundamental misconceptions. The study found that misconceptions were the most frequent source of logic errors, and led to the most difficult errors for students to resolve. The study suggests that targeting these common errors in teaching practices may help to reduce student frustration [29].

A more recent example is a study from 2020, where Emerson *et. al.* attempted to identify novice coding misconceptions in block-based programming. They conducted their research using *PRIME* at a large university in southeastern United States. PRIME is an adaptive block-based programming environment designed to support novices as they learn introductory programming concepts. The target courses were two online sections of an introductory course for undergraduate engineering majors, where a total of 248 participated in the study. They logged on to the PRIME system, allowing the researchers to analyze the programs of the 222 students who attempted at least one activity. They performed a fairly complicated analysis by clustering the student programs through cluster-based algorithms adopted from Bayesian Gaussian Mixture Models[30], which attempt to find the smallest number of clusters that separate the data, in this case, student programs. They identified three distinct clusters; "Exploration", "Disorganized" and "Near miss" programs. They concluded that the ability of identifying these clusters may be an indication of the existence of more general identifiable patterns across block-based programs that can be automatically detected. They continued by stating that identifying these in real-time could lead to a better support system for novice programmers who may otherwise disengage from the assignments [31].

2.4 Summary and Findings

All of the papers mentioned in this section are to some degree relevant to our research, and among these papers there are several methods and techniques applied to answer similar research questions. The difference in methodology makes finding any conclusive evidence difficult, however we can look for statistical relationships that might indicate what types of errors are more common within several papers. It should also be mentioned that there might be even more relevant research papers out there, and this literature review was based on our own protocol described later in section 3.2.

A vast majority of the research is based on compilation errors, and to some extent run-time errors, compared to errors where the program compiles and runs, but it gives the wrong result. The obvious explanation for this is likely that the former type of errors have been much easier to find and categorize since, as it can be collected automatically by tools, surveys or even direct observation by teachers or students in the form of error messages. The "wrong result" type of errors however, are harder to find automatically and needs a more thorough analysis to find out what the cause of the mistake actually was. These types of mistakes can be as valuable or even more so than compilation and run-time errors, because they often represent a students misconception, whereas compilation- and run-time errors often represent sloppy mistakes. One way of attempting to find the "wrong result" type of errors is to conduct interviews with students and/or educators like Hristova and Kaczmarczyk did [12][15].

One could also conduct case studies, where the researchers observe the students while they are writing the code, e.g. having them think out loud or something similar. One example of this is a master thesis written by Johansen from 2015. During the research phase the researchers tasked 23 students of partaking in interviews after they complete the mandatory assignments of a second semester course in object oriented programming using Java. Additionally, the researchers designed some smaller tasks for the research subjects, and observed them while they completed them in order to gather some supplemental data for the study [32]. A similar study was also conducted by Rørnes *et. al.*, where they recruited nine students for semi-structured interviews, asking the students to think out loud and explain how different code snippets would be executed [33]. Although the latter study was more focused towards program comprehension and mental models these are still both good examples of methodology suited for finding the "wrong result" type of errors. We believe that more research investigating these types of mistakes could be highly valuable for educators, students and educational systems in general, and that is why we should consider prioritizing in the future.

Another observation we have made is that most of the studies where student written code is analyzed uses code submitted through assignments from targeted courses or tasks designed by the researchers themselves. These programming tasks all share the fact that the students are given several days to complete their code before final delivery. This would give the students the possibility of multiple

iterations of error handling before final delivery, allowing them to fix mistakes which could have been included in the research. This could potentially give a skewed interpretation of the most common error distribution amongst programmers, leading to inaccurate results and conclusions. One way of avoiding this is to monitor every single compilation throughout the entire coding process for each task. This was done by Brown *et. al.* and Altadmri *et. al.*[9, 19, 21], allowing them to gather all the errors as they were being made throughout the students development phase and not just their final delivery.

A different consequence of code being gathered from assignments spanning over several days, is the availability of different aids in such settings. Students can use modern day IDEs with support for syntax error warnings and auto-completion during the development phase, and with internet available it is very easy to find good code examples similar to their own case online. This would obviously pose a threat to the validity of the research as the data collected could potentially mislead the researchers into finding common errors and misconceptions that does not represent the novice students at all. There are several ways of eliminating this issue, one of them being the previously mentioned methodology of observation of students writing code and another one being analyzing final exam answers instead of multiple day assignments. Veerasamy *et. al.*[24] did exactly this when they conducted their research in 2016. It should be said that this research was conducted on take-home e-exams, allowing the students to use these types of different aids anyways, as they most likely were unsupervised. Additionally, it is easy to cooperate with other people in this type of exam, and the only way of catching the cheaters is through plagiarism checks, which is easy to work around. This poses another threat for the validity of the research, as the code they analyze for one student could have been written by a different person entirely. We believe that gathering data from at-school exams with no aids allowed could potentially give researchers a more accurate representation of students' common misconceptions. It should be mentioned that this approach also has some downsides, particularly the fact that this is a unnatural way of working as a programmer. Professional programmers typically use various aids, and the no-aids situation would typically increase the amount of syntax errors and other "simple" errors, compared to the errors indicating deeper misunderstandings. Perhaps a middle ground of supervised e-exams with a list of relevant syntax and a simple e-exam system offering support like syntax highlighting and automatic indentation could be a good solution.

2.4.1 Research Questions

Through this literature review we have attempted to find potential answers to the research questions presented in chapter 1.1.2. The results found during the literature review will be used as supplementation to the results found during our own research, where we can look for similarities or differences between the two.

In terms of Q1 and Q2 we have observed that the most common mistakes for both Java and Python identified are mainly syntactical. We believe that one of the

main reasons for this is, as previously mentioned, the fact that many studies gather compilation and run-time errors which are often caused by syntactical errors. Identifying common specific error categories is hard, as different researchers tend to have different categories.

Regarding Q3 there is the obvious correlation of them both having a high concentration of syntactical errors. Apart from that, looking at existing literature it seems like misconceptions identified for different programming languages are very language independent. This claim is supported by the study from Caceffo *et. al*[16], where they attempted to convert misconceptions from Python and C to Java. They found that only 10% of misconceptions could be directly mapped from Python to Java, which is quite a small portion.

Finally, in regards to Q4 we could not find any answers through the literature review, as we did not find any similar research in any of the papers.

2.5 Contribution

There are several factors that argues the novelty of our research. One factor lays in the demographic we use in our study. Studies conducted investigating errors and misconceptions in Java and Python amongst novice Norwegian students are scarce at best. Another factor is the fact that we analyze exams, both take-home and regular in-class, taken by students in CS1 and CS2 courses. More often than not the code analyzed in existing studies are written during assignments and given tasks, rather than finals, which could potentially affect the results of the study, as the students tend to focus and prepare more on their finals. The fact that we analyze code from take-home and in-class exams also allows us to compare the results from the two, something we have not found in any of the papers read through our literature review. Additionally, a lot of the code we analyzed was written in Inspera Assessment[34], which is a different tool than the ones we have come across. Inspera Assessment is a generic e-exam system with very limited support for coding. The only help one would get is some syntax highlighting, whereas other features that would also be expected in a programming environment or even rather simple coding editor (e.g., ability to compile and run, debug by stepping through the code, auto-completion of words, etc....) is not provided by Inspera. By discovering the most common errors made amongst Norwegian students, we wish to form recommendations to inform course development and improve teaching practices in Norway, more specifically courses directed at novice programmers in Java and Python, at NTNU, Trondheim.

Chapter 3

Methodology

This section outlines the methodology employed in our research study, designed to answer the research questions regarding common errors encountered by novice programming students at the Norwegian University of Science and Technology (NTNU). Our research questions motivated the need for a comprehensive analysis of a large number of exam submissions, to identify error patterns and understand their causes.

3.1 Research Design

This study aimed to identify errors made by novice programmers at NTNU and sought to understand the underlying challenges faced by students in their programming journey.
Various research methodologies could be applicable to address these questions, including ethnographic studies, surveys, experimental designs, or purely quantitative analyses. However, each of these approaches has its limitations. For instance, an ethnographic study might provide rich context but is time-consuming and difficult to generalize. Surveys can yield broad responses but might miss the nuances of individual coding challenges. Experimental designs, while providing causal evidence, may not be practical in a natural classroom setting. Purely quantitative analyses, on the other hand, would miss the context and the narrative around student errors.

To best answer our research questions, we adopted a mixed-methods approach that combines quantitative and qualitative methods. This approach, using a Python program that automatically logs and categorizes student errors, provides a comprehensive analysis of the most common errors and their patterns over time. Concurrently, qualitative insights help us understand the context and reasons behind these errors. This approach is widely recognized as an effective methodology for examining complex phenomena, as it offers a more comprehensive understanding of the subject matter and facilitates the validation of findings through data triangulation [35–37]. Mixed-methods research has gained prominence in recent years due to

its ability to provide more robust insights and a deeper understanding of the research problem [38]. By employing this approach in our study, we can effectively analyze and interpret the programming errors made by novice programmers while accounting for various contextual factors that might influence their performance [39].

3.1.1 Python Script and Unittest Customization

The Python script and unittests used in this study were meticulously customized to address the specific needs of the research, ensuring accurate identification and categorization of errors made by novice programmers at NTNU. The process began with a thorough examination of the programming concepts and challenges commonly encountered by novice programmers. Based on this assessment, the research team identified the most relevant programming concepts to be tested in the study. Next, a set of unittests was developed for each programming concept, with the tests designed to target common errors made by novice programmers. These unittests were derived from real-world examples of code previously submitted by students, ensuring their relevance and effectiveness.

The Python script was then tailored to incorporate these customized unittests, allowing for seamless and accurate error identification across a large number of exam submissions. Furthermore, the script was designed to be flexible, enabling adjustments to the unittests as needed throughout the study, in order to refine their accuracy and comprehensiveness. This iterative process of selection and refinement of unittests, combined with the customized Python script, ensured that the research methodology was both rigorous and robust. As a result, the study provided valuable insights into the programming errors made by novice programmers

3.1.2 Generalizability

One of the primary limitations of a case study is its limited generalizability. Case studies typically focus on a single or a few instances of a phenomenon, which may provide valuable insights but may not be representative of the broader population [40]. In contrast, the mixed-methods approach used in this study, as advocated by Creswell & Plano Clark[39], allowed for the analysis of a larger sample of student exam submissions. This provided a more comprehensive understanding of the programming errors made by novice programmers at NTNU. This increased the generalizability of the findings and made the results more applicable to a wider range of programming students [41].

3.1.3 Depth and Breadth of Data

While case studies can provide rich, detailed information about specific instances, they may not capture the full range of errors and challenges faced by novice programmers [42]. The mixed-methods approach, as described by Creswell[35],

adopted in this study enabled researchers to collect and analyze data from a large quantity of exam submissions. This allowed for a more comprehensive analysis of the errors made by students, capturing both the depth and breadth of the programming challenges faced by the study population [36].

3.1.4 Efficiency and Resource Allocation

Conducting a case study can be time-consuming and resource-intensive, requiring researchers to collect, analyze, and interpret large amounts of data from a single or few instances. The mixed-methods approach used in this study allowed for more efficient data collection and analysis, as the Python program with unittest framework automated the process of identifying and logging errors made by students. By leveraging the available data and technology, the researchers were able to allocate their time and resources more effectively, focusing on analyzing the data and deriving valuable insights for programming education.

3.1.5 Validity and Credibility

The mixed-methods approach used in this study facilitated data triangulation, enhancing the credibility and validity of the research findings. By using both quantitative and qualitative methods to collect, analyze, and interpret data, researchers were able to cross-validate their findings and draw more robust conclusions about the programming errors made by novice programmers. In contrast, a case study may not provide the same level of data triangulation, limiting the potential for cross-validation and increasing the risk of bias in the research findings.

In conclusion, the mixed-methods approach employed in this study offered several advantages over a case study, including increased generalizability, a more comprehensive understanding of the programming errors made by novice programmers, improved efficiency and resource allocation, and enhanced credibility and validity of the research findings. By using this approach, researchers were able to provide valuable insights for programming education that can inform the development of targeted instructional strategies and interventions to support novice programmers.

3.2 Literature Review

There are several methods and strategies one can employ when performing a literature review. Perhaps the most common among them being narrative, integrative, systematic or meta-analysis, as mentioned by Snyder [43]. We tried, to the best of our ability, to do a systematic review in order to attempt to identify, evaluate and interpret all available relevant research. The reason for this is because the literature study itself was a major part of this project so we wanted to do it thoroughly. We performed the review with a simplified version of the guidelines described by Kitchenham and Charters [44]. The reason we chose to follow their guidelines was because they were specified for Software Engineering and therefore

a good fit. By creating our research questions and a simple review protocol, as seen below, we tried to cover all available relevant research and review it.

Review protocol:

1. Perform a search with a suitable search query on either Oria NTNU or Google Scholar.
2. Read the headings from the resulting articles from the search.
3. Read the abstracts of the articles with related headings.
4. Read the articles with related abstracts.

The initial search gave a lot of hits meaning we had to have some requirements for the headlines to filter out the most relevant articles early on in the process. Initially, any headline, from the predefined search queries, including any form of the words; error, mistake, misconception or misunderstanding made it to the next step of the protocol. However, after realizing that this might cause us to miss a lot of relevant literature, we repeated the search process, adding new articles to the next step of the protocol, solely based on our own judgement.

3.2.1 Finding Papers

To find good papers to include in a literature review, it is important to have a good search query encompassing the most important aspects of the research question. As the intention is to perform research on common programming mistakes made by programmers in Java or Python, it is natural to include the key words common, errors, programming and Java/Python when searching. In addition, as we are looking for novice programmers, we can include words such as beginner or novice. We also intend to research the errors made in an academic computer science course, we need to include key words such as students, computer science, or university to the search query. Combining these key words in to different search queries gives many results containing relevant papers which can be used in the literature review and analysis. Some examples of said search queries can be seen in the list below. In order to come as close to a systematic review as possible we made multiple searches with slight altercations. These altercations could be different synonyms to "error", removing words like "student" and "computer science" from the query entirely or adding new words like "coding". Most searches were made on Google Scholar as we feel like it covers most research in general.

Example search queries:

- "Frequent Java errors made by beginner computer science students"
- "Common Python mistakes made by novice programmers"
- "Novice programming mistakes"

3.3 Data Collection and Analysis

Sample Size and Selection: In this study, a total of 3,637 exam submissions were analyzed, which were collected from three separate instances: an unsupervised home exam in 2021 (H21) with 2,105 submissions, and two supervised on-campus school exams in 2022 (S22) with 1,261 submissions from the morning session and 271 submissions from the evening session. The selection of exam submissions for analysis was based on a stratified random sampling technique to ensure that the sample was representative of the novice programming students at the Norwegian University of Science and Technology (NTNU). The stratification was done according to the different instances (H21, S22 morning, and S22 evening) to account for potential variations in student performance and exam conditions. This sampling method provided a diverse and robust sample of exam submissions, which allowed for a more comprehensive understanding of the programming errors made by novice programmers at NTNU. The large sample size and the careful selection process strengthened the generalizability of the study findings, contributing to the validity and credibility of the research results. Further information regarding the specific exams, their corresponding tasks, and the suggested solutions can be found in the appendices of this document. The appendices provide an in-depth overview of the problems students were asked to solve during these examinations, offering readers additional context for understanding the error analysis. By examining the tasks and the proposed solutions, readers can gain a deeper comprehension of the complexity and the requirements of the exam submissions, thus contributing to a more nuanced understanding of the novice programming students' challenges at the Norwegian University of Science and Technology (NTNU).

Exam Instance	Submissions	Location	Duration	Tasks	Aids
H21	2,105	At home	4 hours	11	All aids available
S22 Morning	1,261	On-campus	4 hours	3	Inspera browser
S22 Evening	271	On-campus	4 hours	3	Inspera browser

Table 3.1: Sample size and selection of exam submissions for analysis

3.3.1 Comparing Java Results

In order to provide a comprehensive understanding of the common programming errors and difficulties faced by students, the results from a separate study on Java exam submissions at the Norwegian University of Science and Technology (NTNU) will be used as a point of comparison and to supplement the findings of this study. This Java study, titled "Common mistakes made by novice programmers in Java" (Bjerkeset, Cook, 2023), was conducted by the same authors, Ole-Christian Bjerkeset and Sigmund William Cook, under the supervision of Guttorm Sindre. The study focused on a manual review of 20 Java exam submissions, examining the frequency of various error categories and providing valuable insights into the

challenges faced by novice Java programmers at NTNU.

By comparing the error frequencies and categories observed in the Python scripts from this study with the results obtained for Java submissions, we can gain valuable insights into the general programming difficulties faced by students, regardless of the programming language used. This integration allows us to identify commonalities and differences in the types of errors and challenges experienced by students across different programming languages.

This comparative analysis can potentially reveal patterns and trends that may be language-specific or universally present across programming languages. Furthermore, the inclusion of Java results in the methodology chapter can help in identifying areas where targeted interventions and support might be needed to assist students in overcoming their programming challenges, thereby contributing to their overall learning experience and success in computer science courses.

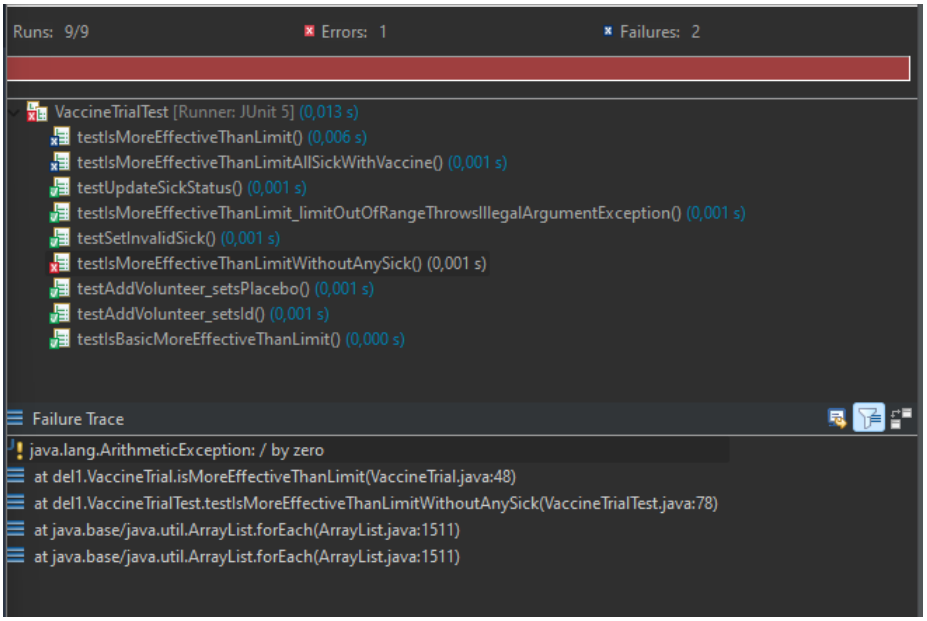


Figure 3.1: Example of a test case

```
public boolean isMoreEffectiveThanLimit(double limit) {
    if (limit < 0 || limit > 1) {
        throw new IllegalArgumentException("Limit should be between 0 and 1");
    }
    int sickAndVaccinated = (int) sickVolunteers.stream().filter(x -> vaccinatedVolunteers.contains(x)).count();
    return 1 - (sickAndVaccinated / vaccinatedVolunteers.size()) > limit;
}
```

Figure 3.2: Example of code

In the Java study conducted by the same authors[1], a code example was provided to illustrate common errors encountered by students (see Figure 2). In this example, an error was found in the test testIsMoreEffectiveThanLimitWithoutAnySick(), and incorrect output occurred in

testIsMoreEffectiveThanLimit() and testIsMoreEffectiveThanLimitAllSickWithVaccine(). Upon selecting the error, the message "ArithmeticException / by zero" was displayed, indicating that the code attempts to divide by zero within the isMoreEffectiveThanLimit() function, which results in an undefined value and an error. Further examination of the code revealed that the root cause of the error was the incorrect usage of types. The candidate divided an int value by a double value, which produced an unexpected value when calculating the output. Fixing this error by changing the value to double resolved the issue, but did not address the incorrect logic used when calculating vaccine efficiency. According to the task instructions, the candidate should have returned the effectiveness based on the formula "1 - (number of people that received the vaccine and got sick / number of people that got sick)". In this example, the student encountered difficulties in understanding and applying the correct types and logic, which aligns with the findings of common errors identified in the larger Java study.

3.3.2 Manual Analysis of Python

In order to gain a deeper understanding of the common errors and misconceptions among novice programmers in Python, we will employ the same manual analysis method used in the Java study [1]. In the Java study, the authors analyzed the exam results of 20 students studying the course TDT4100 at the Norwegian University of Science and Technology (NTNU). These exams were written in Java and were graded by the lecturers. The authors identified the most common errors made by the students in their exam answers, compiled a list of these errors and their respective frequencies.

The authors of the Java study also conducted a manual review of the first task in 20 exam submissions that contained errors. These submissions were selected randomly. The data contained exam submissions from students with the project files, as well as a series of tests which had been performed on the students' code. These tests checked the code for the correct output and outputted an error message or exception if the code was incorrect. The areas where the tests failed or ran into an error were manually reviewed, and all errors, including submissions where multiple errors occurred, were categorized based on the errors found.

Following this similar approach in our study, we hope to provide comparable insights into the common errors made by students learning Python. This will allow us to better compare the findings from both programming languages and assess any similarities or differences in the challenges faced by new programmers.

Similar to the Java study, we will carefully review each Python submission to identify errors and classify them according to their nature and severity. Examples of errors we will be looking for include syntax errors, logic errors, incorrect use of data types, and issues with control structures. During the analysis, we will pay close attention to cases where multiple errors might be present, as well as the underlying misconceptions that may have led to the errors.

Please note that this data is not meant to represent the overall performance of

students in these exams, but rather to shed light on common error patterns and their prevalence. Through this analysis, we aim to identify common error patterns, which can guide future teaching strategies and inform students about potential pitfalls to avoid during programming tasks. Furthermore, this analysis can aid in the development of educational resources and support materials, with a focus on addressing the most frequently encountered errors.

It is important to note that when running a Python program, the code execution halts at the first instance of an error. Consequently, in situations where multiple errors are encountered, the error message stops the program and only the first error is recorded in the log. This means that a student could have mostly correct code, but a simple typing mistake can result in the same error message as that of a student who has misunderstood the programming concept as a whole. By paying close attention to multiple errors and misconceptions during the analysis, we can gain a more comprehensive understanding of the challenges faced by novice programmers.

The manual analysis for this study was conducted on a carefully chosen subset of Python exam submissions from NTNU students. The data used in the analysis, as depicted in Table 4.11, was collected from a sample of examination submissions from three distinct NTNU exams that were held in the autumn semesters of 2021 and 2022. Notably, the 2021 exam was a home exam, while the 2022 exams were school exams.

To address the selection process more specifically, it's important to note that while the selection incorporated elements of randomness, it was not entirely random. It was recognized early in the process that selecting an excessively weak or excessively strong student's submissions would not provide valuable insights for the purpose of this study. Therefore, submissions that were almost blank or almost entirely correct were deemed unsuitable and excluded from the selection pool.

Instead, the selection process aimed to sample a representation of students who demonstrated a range of abilities, but who also exhibited a meaningful number of errors. This approach allowed for a more targeted and insightful examination of the types of errors that novice programmers tend to make. Consequently, while the selection process did involve random elements, it was constrained by the pre-established condition that the selected submissions should be of substantive interest and value to the study.

The total number of submissions sampled per exam set were as follows: 33 submissions were analyzed from the H21 exam set, and 9 submissions each from the S22Morgen and S22Kveld exam sets. This totals to 51 submissions across the three exam sets. The errors in these submissions were then categorized and quantified, with the results presented in Table 4.11, 4.12 and 4.13.

Through this analysis, we aim to identify common error patterns, which can guide future teaching strategies and inform students about potential pitfalls to avoid during programming tasks. Furthermore, this analysis can aid in the development of educational resources and support materials, with a focus on addressing the most frequently encountered errors. One example of a Python code error identified

during the manual analysis is the following:

Code listing 3.1: Python example with errors

```
def alfa(streng, tall):
    if int != 0:
        return streng.max()
    else:
        return streng.min()
```

This code contains multiple errors, which we can break down in detail as follows:

1. In the conditional statement `if int != 0`, the name `int` actually refers to a built-in Python function for converting values to integers. The condition should instead check if the variable `tall` is not equal to zero. The correct statement would be `if tall != 0`.
2. The `streng.max()` and `streng.min()` methods do not exist for strings in Python. Instead, the programmer likely intended to find the maximum or minimum character in the string based on their Unicode code point. To achieve this, the built-in Python functions `max()` and `min()` should be used. The correct statements would be `return max(streng)` and `return min(streng)`.

After identifying and correcting these errors, the fixed code would look like this:

Code listing 3.2: Corrected Python example

```
def alfa(streng, tall):
    if tall != 0:
        return max(streng)
    else:
        return min(streng)
```

During the analysis, we will take note of the frequency and prevalence of each error type, as well as any patterns or trends that emerge in the students' code. The findings from this manual analysis will then be compared to the results obtained from the Java study to determine if there are any commonalities in the types of errors encountered by novice programmers, regardless of the programming language being used.

By using the same manual analysis method for both studies, we aim to ensure a consistent and reliable approach to identifying and understanding the challenges faced by new programmers. This will provide valuable insights for educators and curriculum developers, enabling them to design more effective learning resources and support for novice programmers.

3.3.3 Data Visualization

Data Visualization: In our study, we opted for tables and pie charts to effectively communicate the findings and help readers better understand the patterns and trends in programming errors made by novice programmers. Tables presented

a clear and organized way to display the data, allowing for easy comparison of error types and frequencies across different programming concepts. Pie charts, on the other hand, provided a visual representation of the proportion of each error type within the overall dataset, enabling readers to quickly grasp the relative prevalence of specific errors made by students. By incorporating tables and pie charts into our study, we successfully conveyed the insights gained from our research, emphasizing the importance of targeted instructional strategies in addressing the challenges faced by novice programmers.

Python Program Execution and Log File Generation

The Python program, designed with unittests, was executed on each of the Python files submitted by the students. This automated process ensured consistent and unbiased error identification across all submissions. Upon execution, the program generated a log file containing the results of the unittesting process for each submission. These log files served as the primary data source for both the qualitative and quantitative analyses in the study.

The code and analysis scripts provided in the GitHub repository `https://github.com/sigmundcook/Master_results` are specifically tailored for the exams under investigation in this study. They are not designed as a general tool and are not directly applicable to other exams or programming assessments. The customized nature of the code enables a thorough examination of the unique challenges faced by novice programmers in the context of these specific exams.

Error Identification and Categorization

The log files were analyzed to identify the errors made by novice programmers during the exam. Errors were categorized using the categories assigned by Pythons unittesting library. This categorization enabled researchers to analyze the data more effectively and identify patterns and trends in the types of errors made by students. Moreover, it provided a systematic framework for understanding the challenges faced by novice programmers and informed the development of targeted instructional strategies to address these issues.

3.3.4 Categorising Data

When the data is collected we will categorize the data. Python's unittesting categorises errors by default. Categorizing data using Python's unittest categories can provide a number of benefits in terms of identifying and addressing common coding issues, as well as providing targeted feedback and instruction to students. Here are some reasons why it is good to categorize data using Python's unittest categories:

- **Provides a standardized framework:** Utilizing Python's unittest categories for error categorization provides educators and programmers with a standardized framework. This approach ensures consistency in identifying and addressing common coding issues.

- **Facilitates targeted instruction:** Categorizing errors allows educators and programmers to identify common issues that students encounter. This information can be used to provide targeted instruction and resources to help students understand programming concepts and improve their overall performance.
- **Enables efficient debugging:** Categorizing errors also helps streamline the debugging process. Programmers can quickly identify the type of error that has occurred, reducing the time and effort required for troubleshooting and making corrections.
- **Supports continuous improvement:** Analyzing the types and frequency of errors in programming tasks enables educators and programmers to identify areas where students may be struggling. This knowledge can be used to adjust the curriculum or instruction, supporting continuous improvement and ensuring the continued relevance and effectiveness of programming education.

Overall, categorizing data using Python's unittest categories can help to streamline the programming education process, provide targeted instruction, and support continuous improvement. By identifying and addressing common coding issues, students can gain a deeper understanding of programming concepts and become more confident and capable programmers.

To better understand the different types of errors that can occur in Python, let's delve into some common categories, detailing the circumstances under which each error might be raised:

- AssertionError occurs when an assertion made in the code fails. Assertions are statements in the code that are used to test for a specified condition. Assertions are commonly used to test preconditions, postconditions, and invariants in the code. For example, an assertion might be used to test that a variable is within a certain range or that a function returns a specific value.
- TypeError occurs when the code attempts to perform an operation on incompatible data types or when using an unsupported operator. For example, trying to multiply a string and an integer, or attempting to use the append method on an integer object will raise a TypeError.
- ValueError occurs when a function or method is passed an argument of the correct type but with an inappropriate value. For example, passing a negative number to a function that only accepts positive integers will raise a ValueError.
- AttributeError occurs when the code attempts to access an attribute of an object that does not exist. For example, if a variable is not an instance of a class and an attempt is made to access one of its attributes, an AttributeError will be raised.
- IndexError occurs when the code attempts to access an index in a sequence (e.g., a list or string) that is out of range. For example, if a list contains only three elements and the code attempts to access the fourth element, an IndexError will be raised.

- KeyError occurs when the code attempts to access a dictionary key that does not exist. For example, if the code attempts to access a value in a dictionary using a key that is not present in the dictionary, a KeyError will be raised.
- NameError occurs when the code attempts to reference a variable or function that does not exist or has not been defined. For example, if the code attempts to use a variable that has not been assigned a value or a function that has not been defined, a NameError will be raised.
- SyntaxError occurs when the code violates the rules of the Python syntax, such as missing a colon or parenthesis. For example, if a colon is omitted in a function definition, a SyntaxError will be raised.
- IndentationError occurs when the code contains incorrect indentation or spacing. In Python, whitespace is used to indicate blocks of code, and incorrect indentation can cause errors. For example, if a line of code is not properly aligned with the rest of the block, an IndentationError will be raised.
- ImportError occurs when the code attempts to import a module that does not exist or cannot be found. For example, if the code attempts to import a module that has not been installed or is not in the current working directory, an ImportError will be raised.

3.3.5 Data Analysis

The data collected from the log files was analyzed to identify the errors made by novice programmers at NTNU. The errors were categorized by programming concept, and the frequency of each error was calculated. The data was also analyzed to identify any patterns or trends in the types of errors made by novice programmers.

Quantitative Analysis

The cornerstone of the quantitative analysis was the application of descriptive statistics to present a detailed account of the error occurrences. The data from Python's unittest framework served as the primary dataset for this exploration. This statistical approach facilitated the clear summarization of the error frequency and distribution across the various categories in the dataset. Not only did this allow for an understandable representation of the information, but it also highlighted patterns and trends in the numerical data that could otherwise remain obscure.

By scrutinizing the frequency of each error category, researchers could pinpoint the recurring errors made by novice programmers. These findings provide an invaluable insight into the areas where new programmers typically face difficulties. Moreover, by comparing these frequencies between the home exam of 2021 and the regular exams of 2022, significant differences in the types of errors made in each scenario were exposed. This comparative approach offers a more dynamic understanding of the learning curve and possible effect of the exam environment on the students' performance.

3.3.6 Qualitative Analysis

Complementing the quantitative findings, qualitative analysis was deployed to add depth to the understanding of the types of errors made by novice programmers. The qualitative approach aimed to unravel the hidden themes and recurring patterns that might be missed by a purely numerical analysis. The raw data for this examination were the log files, containing details of every error occurrence.

The analysis began with a thorough review of these logs, involving a manual search for patterns and trends. This process demanded meticulous attention to detail, as the objective was not only to identify the errors but also to discern any underlying issues that might be leading to these errors.

An example of this is that among the major themes that emerged, we found a consistent struggle with syntax errors among novice programmers. This suggests that many students may not be fully familiar with the language's syntactic rules, highlighting a potential area for increased instructional focus. Furthermore, another common struggle was related to the use of loops and conditional statements. This could be indicative of a challenge in grasping fundamental programming logic and could signal a need for further pedagogical emphasis on these concepts.

In sum, this multifaceted approach of employing both quantitative and qualitative analysis methods allowed for a more comprehensive and nuanced understanding of the difficulties novice programmers face while learning Python. This synthesis of findings stands to inform and enhance educational strategies in programming instruction.

3.3.7 Python Script Development

The Python script used in this study was developed to analyze the exam submissions from students who study programming at NTNU. The script was designed to extract the code from a large CSV sheet, run unittests on the code, and log the results of the unittests to a CSV file containing the students' errors.

The first step in developing the script was to identify the key programming concepts that novice programmers at NTNU typically struggle with. Using this information, the research team then developed a set of unit tests for each programming concept. The unit tests were designed to identify common errors made by novice programmers and were based on real-world examples of code that had been previously submitted by students.

Once the script was developed, it was used to analyze the exam submissions from students who study programming at NTNU. The script extracted the code from the CSV sheet and ran the set of unit tests on the code. The results of the unittests were then logged to a CSV file containing the students' errors. The CSV file was designed to be easily readable, with each row representing a single error and columns indicating the programming concept and specific error that was made.

3.3.8 Ethical Considerations

The principal ethical consideration in this study pertains to the protection of participants' privacy. To uphold this, all data employed in this study was already anonymized prior to the authors receiving it, ensuring the integrity of the privacy of the participants. The researchers never had access to any identifiable features, such as names, student IDs, or emails. The Python program was designed to only collect data on the errors made by the participants, and no personally identifiable information was collected or stored.

The anonymity of the participants is crucial to protect their privacy and to ensure that the study is conducted in an ethical manner. The data collected from the log files was analyzed in a way that ensures that individual participants could not be identified. This means that the data collected in this study was combined and presented in a way that did not allow for individual participants to be identified. Instead of looking at individual student data, the researchers combined the data so that it shows the overall patterns and trends in the errors made by novice programmers.

This approach helps protect the privacy of the participants and ensures that their personal information is not shared or used in a way that could potentially identify them. Additionally, presenting data in an aggregated form can be useful in identifying broad patterns and trends, which can be valuable in developing effective teaching and learning strategies for programming courses at NTNU.

3.3.9 Conclusion

In conclusion, the methodology used in this study allowed for the development of a Python script that automatically unittested a large number of exam submissions from novice programming students at NTNU. The results of the study provided valuable insights into the common errors made by the students and their overall performance. The script developed in this study can be used as a tool for automating the assessment of programming assignments, which can save time and resources for instructors.

Chapter 4

Results and Findings

In this results section, we present data on the frequency of errors made by students in programming exams. Our analysis focuses on the 24 most frequent errors, presented in a table. However, we also consider the total number of errors within each category, including those that occur less frequently.

We begin by providing an overview of the errors made by students and the frequency with which they occurred. We present a table that lists the 24 most common errors, along with the number of times each error was made by students. These errors are categorized by type, such as index errors, assertion errors, type errors, name errors, among others.

Finally, we consider the total number of errors within each category, including those that occur less frequently. By doing so, we can gain a more complete understanding of the distribution of errors and identify any additional trends or patterns that may be relevant to improving programming education.

Overall, this results section provides a comprehensive analysis of the frequency and types of errors made by students in programming exams. The information presented will be useful for identifying areas where students may be struggling and for tailoring programming education to better address their needs.

4.1 Summarising Errors

Tables 4.1, 4.2, and 4.3 provide an overview of the unique errors encountered in the Home exam autumn 2021, Morning School exam autumn 2022, and Evening School exam autumn 2022, respectively.

Unique Errors Home exam autumn 2021	count
NameError("name 'oppdater_matvare' is not def...	669
NameError("name 'finn_pris' is not defined")]]	479
AssertionError('0 != 99')]]	284
IndexError('list index out of range')]]	266
IndexError('string index out of range')]]	231
AssertionError('"odde' != 'des'- odde+ de...	146
AssertionError('"des' != 'odde'- des+ odd...	145
TypeError('">' not supported between instance...	110
AssertionError('None != 0')]]	106
AttributeError('"int' object has no attribute...	103
AssertionError('59 != 0')]]	86
AssertionError('1 != 2')]]	80
AssertionError('None != 1')]]	80
NameError("name 'vare' is not defined")]]	79
AssertionError('None != []')]]	71
AssertionError('"1' != 1")]]	62
TypeError("unsupported operand type(s) for -:...	52
NameError("name 'vis_priser' is not defined")]]	51
TypeError('"int' object is not iterable")]]	50
NameError("name 'oppdater_beholdning' is not ...	43
TypeError('"int' object is not subscriptable")]]	43
NameError("name 'matvare' is not defined")]]	42
NameError("name 'matvarer' is not defined")]]	37
AssertionError('"0' != 99")]]	35

Table 4.1: Counts of Unique Errors

4.1.1 Interpretation of Results Home Exam Autumn 2021

Table 4.1 provides a breakdown of the unique errors encountered during the Home Exam in Autumn 2021, indicating the frequency of each error type.

- The most common error is `NameError("name 'oppdater_matvare' is not defined")`, with 669 instances. It's important to note that the high frequency of this error could be inflated. The error indicates that the function or variable named `oppdater_matvare` was frequently referenced but was not defined or imported correctly. This error could arise from the task's setup, which allows students to reference another function that may not have been correctly imported into the unittest environment. It's also worth noting that a contributing factor could be the typical practice in exams where there is a progression of subtasks where subsequent tasks build on the code from previous tasks. In this case, the question states that students may assume that a function sought in the previous task works as specified, even if they haven't successfully solved that task. This is done to avoid follow-up errors, for instance, when a

student who fails to solve 3(a) but would be capable of solving 3(b) and 3(c) using a function from 3(a), still has an opportunity to accomplish these tasks instead of being penalized too harshly for failing a single subtask. Therefore, there could be students who did not solve the function from the previous task but were still allowed to solve the current function and would receive a score for it. An example of this can be the following. The task 3.3, as described in the exam, required the students to update the inventory of a store given a list of changes in the format `[[item_name, quantity],...]`. Here is a suggested solution to the problem:

```
def oppdater_beholdning(beholdning, endringer):
    for endring in endringer:
        item, quantity = endring
        beholdning[item] += quantity
    return beholdning
```

This code iterates over the list of changes, unpacks each change into an item name and a quantity, and updates the corresponding item's quantity in the inventory. The most common error found in this task was the NameError stating "name 'oppdater_matvare' is not defined".
However, this correct code would produce the NameError, because the function `oppdater_matvare` is not imported into the unittest script. In order for the unittest to use this function, the code must define it within the script itself or correctly import it if it's defined elsewhere. The high frequency of this error can be attributed to the task's requirements and the exam environment's setup, where the student could assume that functions written in a seperate task were allowed to be used in another task.

- The second most frequent error, `NameError("name 'finn_pris' is not defined")`, follows a similar trend to the prior example. It also appears to be affected by the setup of the task and the testing environment, as it occurred 479 times.
- Various types of `AssertionError` are also quite common, signifying that there are several instances where conditions assumed to be true in the code were not met. This type of error is usually found when assert statements are used for debugging. The discrepancies between expected and actual values during the execution of the code hint at potential misunderstandings about the expected program behavior.
- The `IndexError` often occurs when there is an attempt to access an index that is out of range in a list or a string. High frequencies of this error, such as `IndexError('list index out of range')` with 266 instances and `IndexError('string index out of range')` with 231 instances, suggest that students may have struggled with managing index boundaries in data structures.|
- Multiple instances of `TypeError` suggest that students applied operations or functions to objects of inappropriate types. This could indicate confusion about type compatibility and the correct use of certain functions and operations.

Unique errors morning School exam autumn 2022	count
AssertionError('False is not true : False')]]	554
IndexError('list index out of range')]]	50
AssertionError('False is not true : True')]]	47
NameError("name 'lenght' is not defined")]]	43
NameError("name 'np' is not defined")]]	41
TypeError("'int' object is not iterable")]]	38
ValueError('The truth value of an array with ...	34
IndexError('index 4 is out of bounds for axis...	33
IndexError('arrays used as indices must be of...	33
AssertionError('None is not true : False')]]	30
AssertionError('None is not true : True')]]	24
IndexError('index 5 is out of bounds for axis...	22
AssertionError('0 != 11')]]	20
AssertionError('3 != 11')]]	19
IndexError('pop index out of range')]]	19
NameError("name 'length' is not defined")]]	17
IndexError('index 4 is out of bounds for axis...	17
AssertionError('0 is not true : 3')]]	17
AssertionError('None is not true : 3')]]	16
AssertionError('4 != 11')]]	14
AssertionError('None != 11')]]	12
AssertionError('7 != 11')]]	12
AssertionError('6 != 11')]]	11
TypeError("'NoneType' object is not iterable")]]	11

Table 4.2: Counts of Unique Errors

4.1.2 Interpretation of Results Morning School Exam Autumn 2022

Table 4.2 enumerates the unique errors encountered during the Morning School Exam in Autumn 2022, with details of each error's frequency.

- The most frequent error was `AssertionError('False is not true : False')`, with 554 occurrences. This error suggests that a certain condition was expected to be true in the student's code but evaluated to false. The frequency of this error could indicate a general misunderstanding of Python's boolean logic or that a common question in the exam was frequently misunderstood.
- `IndexError('list index out of range')` occurred 50 times, which shows that a common problem was mishandling of list indices. This error is raised when a non-existent index is accessed in a list.
- `NameError("name 'lenght' is not defined")` and `NameError("name 'length' is not defined")]` were other common errors, with 43 and 17 instances respectively. These errors suggest that students may have attempted to use a variable or function that wasn't defined or was misspelled. The high frequency

of these errors indicates a potential difficulty in understanding or applying Python's syntax and variable naming conventions.

- With 41 instances, `NameError("name 'np' is not defined")` signifies that many students were trying to use the numpy library without correctly importing it.
- `TypeError("'int' object is not iterable")` occurred 38 times. This error often arises when there is an attempt to iterate over an integer, which isn't possible since integers are not iterable objects. This suggests that there might be some misunderstanding regarding the data types that are iterable in Python.
- The high occurrence of `ValueError('The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()')` and `IndexError('arrays used as indices must be of integer (or boolean) type'` indicate that students struggled with correct use of arrays, specifically numpy arrays. The former error points towards an incorrect attempt to check the truth value of an entire array, and the latter indicates misuse of arrays as indices.
- `AssertionError('None is not true : False')` and other assertion errors like `AssertionError('0 != 11')` indicate that students' functions are not returning the expected results.
- Lastly, `TypeError("'NoneType' object is not iterable")` suggests that the students attempted to iterate over a NoneType object, which isn't iterable. This could indicate an issue where students might have failed to handle default or exceptional cases where a function could return None.

Unique Errors evening School exam autumn 2022	count
IndexError('arrays used as indices must be of...	11
AssertionError('0 != 15')]]	10
IndexError('list index out of range')]]	9
TypeError('only integer scalar arrays can be ...	7
ValueError('The truth value of an array with ...	6
AssertionError('None is not true : M3')]]	6
TypeError("'int' object is not subscriptable")]]	6
TypeError("'int' object is not iterable")]]	6
IndexError('index 4 is out of bounds for axis...	5
NameError("name 'M3' is not defined")]]	4
AssertionError('0 is not true : 38.2')]]	4
TypeError('only size-1 arrays can be converte...	4
IndexError('index 4 is out of bounds for axis...	4
TypeError("object of type 'int' has no len()")]]	4
NameError("name 'i' is not defined")]]	4
TypeError("'tuple' object is not callable")]]	3
NameError("name 'true' is not defined")]]	3
AssertionError('None is not true : 38.2')]]	3
AssertionError('None != 15')]]	3
AttributeError("'numpy.float64' object has no...	2
UnboundLocalError("local variable 'sum' refer...	2
TypeError("unhashable type: 'list'")]]	2
TimeoutError()]]	2
AssertionError('8 != 15')]]	2

Table 4.3: Counts of Unique Errors

4.1.3 Interpretation of Results Evening School Exam Autumn 2022

Table 4.3 presents the unique errors encountered during the Evening School Exam in Autumn 2022, detailing the frequency of each error type.

- The most frequent error was `IndexError('arrays used as indices must be of integer (or boolean) type')`, with 11 occurrences. This error suggests that the students might have used arrays as indices that weren't of integer or boolean type. This indicates a common misunderstanding regarding the usage of array indices in Python, specifically in the numpy library.

  ```
  import numpy as np
  A = np.array([1, 2, 3])
  indices = np.array([0.5, 1.5, 2.5])
  print(A[indices])
  ```

  In the example above, using a numpy array of float indices to index another numpy array would result in the `IndexError`. Students need to ensure that

they use only integer or boolean values for indexing arrays.

- The second most frequent error, `AssertionError('0 != 15')`, and other assertion errors indicate that the students' functions didn't return the expected results. For example, the `AssertionError('0 != 15')` error suggests that the output of a student's function was 0 when the expected result was 15.
- Similar to the Home exam, `IndexError('list index out of range')` is also prevalent in the Evening exam, with 9 instances. This error often suggests a misunderstanding of how indexing works in Python, specifically the handling of index boundaries in data structures.
- Multiple instances of `TypeError`, like `TypeError(only integer scalar arrays can be converted to a scalar index')` and `TypeError("'int' object is not subscriptable")]` indicate that students were not using the correct data types for certain operations or functions. This points towards potential gaps in understanding Python's type system and the compatibility of different data types with different operations and functions.
- The error `ValueError('The truth value of an array with ...` indicates a misunderstanding about the evaluation of arrays in boolean contexts. Students seem to have been checking the truth value of an entire array, which is ambiguous, instead of checking the truth values of the individual elements or using numpy array methods like `numpy.all()` or `numpy.any()`.
- Lastly, `NameError` like `NameError("name 'M3' is not defined")]` and `NameError("name 'i' is not defined")]` suggest that there were instances where students attempted to use variables or functions that had not been defined in the scope of their use.

4.1.4 Comparative Analysis of Errors Across Exams

In this section, we compare and contrast the nature and frequency of errors across the Home exam Autumn 2021, the Evening School exam Autumn 2022, and the Morning School exam Autumn 2022.

- The most common error in both the Home exam and the Morning School exam was related to assertions. In the Home exam, students often encountered errors when their code did not align with the expected results in the unittest environment. This is due to incorrect logic or misunderstanding of the task. This pattern was observed again in the Morning School exam with `AssertionError('False is not true : False')`. This error suggests that students had a misunderstanding about Python's boolean logic or the requirements of the question. On the other hand, the most common error in the Evening School exam involved incorrect handling of array indices.
- Both the Home exam and the Evening School exam frequently reported NameError, which was commonly due to either the specific setup of the exams or students' misunderstanding of Python's scope rules for variables and functions. Students frequently referenced functions or variables that weren't appropriately defined or imported. While this error was less prominent

in the Morning School exam, the discrepancy may be attributed to the exam structure of the Home exam in 2021, where students were allowed to reference functions written in previous tasks. This could potentially inflate the number of NameErrors as a correct answer might still raise an error if the testing environment does not have access to the referenced function. Thus, the high incidence of NameErrors in the Home exam may not entirely reflect students' misunderstanding, but instead could be partially due to limitations in the testing environment's setup.

- `TypeError` and `ValueError` were common across all three exams, often caused by misuse of data types and data structures. The prevalence of these errors indicates a potential struggle among students in understanding and applying Python's type system, as well as confusion about the appropriate use of various Python functions and methods.
- In both School exams, students encountered issues with handling array indices, leading to `IndexError`. The high frequency of this error suggests that managing index boundaries in data structures may be a common challenge.
- The Home exam and the Morning School exam saw errors related to an undefined variable named 'length' or 'lenght', suggesting difficulties in managing variable names and possibly, typos.
- The Morning School exam also highlighted the students' struggles with correctly importing and using the numpy library, as indicated by the error `NameError("name 'np' is not defined")`.

In conclusion, the unique error breakdown of these exams helps identify areas of common difficulty among students, which include handling Python's type system, understanding boolean logic, managing index boundaries, and correctly defining and using variables and functions. There are also issues related to the correct import and use of external libraries like numpy. This analysis provides valuable insights that could be used to better structure and focus Python learning materials and instruction.

Sum of error categories Home exam autumn 2021	**count**
NameError	1853
AssertionError	1724
TypeError	611
IndexError	510
AttributeError	232
UnboundLocalError	57
ValueError	83
KeyError	45

Table 4.4: Caption

Sum of Error Categories morning School exam autumn 2022	count
AssertionError	814
NameError	240
IndexError	181
TypeError	149
ValueError	46
AttributeError	41
UnboundLocalError	31
KeyError	0

Table 4.5: Sum of Error Categories

Sum of Error Categories evening School exam autumn 2022	count
TypeError	44
NameError	38
AssertionError	36
IndexError	31
AttributeError	10
UnboundLocalError	8
ValueError	7
KeyError	0

Table 4.6: Sum of Error Categories

4.2 Proportions of Error Categories in Different Exams

Error Category	Home Exam 2021	School Exam Morning 2022	School Exam Evening 2022
NameError	1853 (36.4%)	240 (16.0%)	38 (21.8%)
AssertionError	1724 (33.8%)	814 (54.2%)	36 (20.7%)
TypeError	611 (12.0%)	149 (9.9%)	44 (25.3%)
IndexError	510 (10.0%)	181 (12.1%)	31 (17.8%)
AttributeError	232 (4.6%)	41 (2.7%)	10 (5.7%)
UnboundLocalError	57 (1.1%)	31 (2.1%)	8 (4.6%)
ValueError	83 (1.6%)	46 (3.1%)	7 (4.0%)
KeyError	45 (0.9%)	0 (0%)	0 (0%)
Total Errors	5095	1502	174

Table 4.7: Error Categories Proportions in Exams

4.2.1 Error Category Proportions in Different Exams

The table 4.7 provides a detailed breakdown of error categories and their proportions in the Home Exam of 2021, and the Morning and Evening School Exams of 2022.

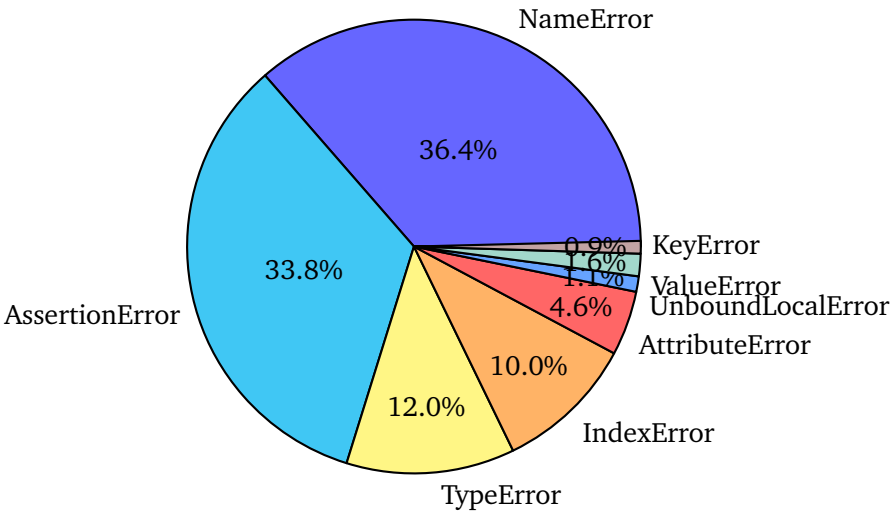


Figure 4.1: Home Exam 2021

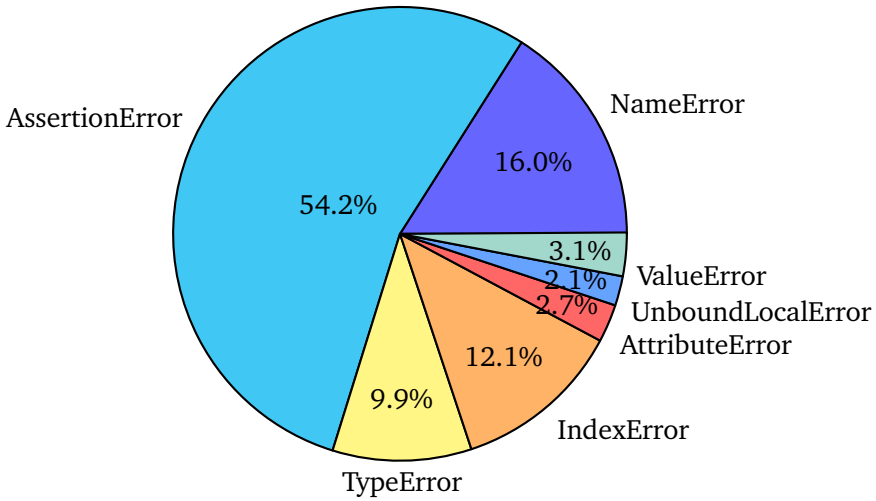


Figure 4.2: School Exam Morning 2022

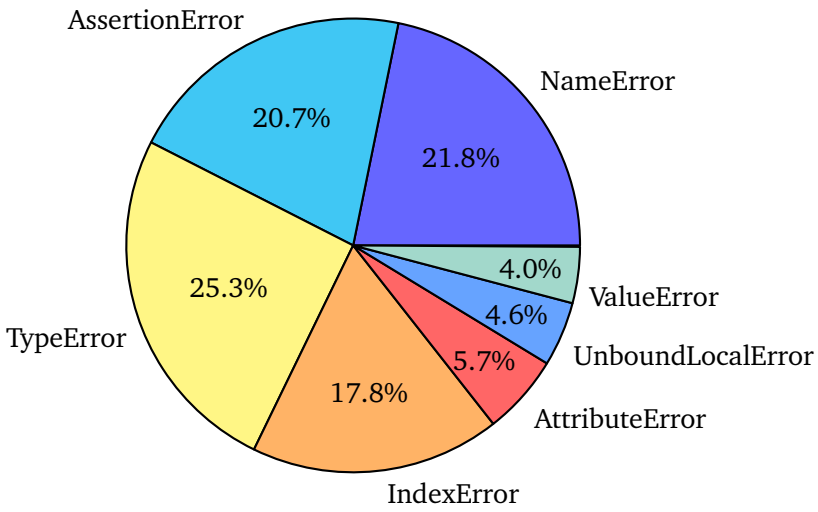


Figure 4.3: School Exam Evening 2022

- **NameError** was most prevalent in the Home exam, constituting 36.4% of all errors. This proportion was significantly higher compared to the Morning School exam (16.0%) and the Evening School exam (21.8%). This could be largely attributed to the structure of the Home exam, which allowed students to reference previously defined functions in their solutions. If the testing environment did not have access to these referenced functions, a NameError was raised, likely leading to an inflated count in this category.
- **AssertionError** was most common in the Morning School exam, making up over half of all errors (54.2%). This error was also prominent in the Home exam (33.8%), but was less frequent in the Evening School exam (20.7%). The high proportion of AssertionError in the Morning School exam might indicate that students had a greater difficulty understanding the requirements and constraints of the tasks, or that other errors were more prevalent, causing the code to stop running.
- **TypeError** and **IndexError** were present across all three exams, but their frequencies varied. TypeError was most prevalent in the Evening School exam (25.%), followed by the Home exam (12.0%) and the Morning School exam (9.9%). IndexError, on the other hand, was more common in the Morning School exam (12.1%), compared to the Home exam (10.0%) and the Evening School exam (17.8%). These errors often stem from incorrect usage of data types and operations or inappropriate handling of data structures, suggesting that these areas could be challenging for students.
- **AttributeError**, **UnboundLocalError**, and **ValueError** made up smaller proportions of the total errors in all three exams. Their occurrences could indicate misconceptions or misunderstandings about object properties, local/global scope, and function parameters, respectively.
- **KeyError** was unique to the Home exam, where it constituted a minor proportion

of total errors (0.9%). This error did not occur in either the Morning or Evening School exams, suggesting that it might be related to the specific tasks in the Home exam that involved handling dictionaries.

Task	Errors	Percentage of students made an error on this task
Oppgave 2a	269	12.80%
Oppgave 2b	131	6.23%
Oppgave 2c	639	30.41%
Oppgave 2g	310	14.75%
Oppgave 2h	637	30.32%
Oppgave 2i	514	24.46%
Oppgave 3.1	774	36.84%
Oppgave 3.2	15	0.71%
Oppgave 3.3	838	39.88%
Oppgave 3.4	538	25.61%
Oppgave 3.5	475	22.61%

Table 4.8: Sum of errors and percentage of students that made an error for each task home exam 2021

Task	Errors	Percentage of students made an error on this task
Oppgave 1	372	29.49%
Oppgave 2	782	61.97%
Oppgave 3	353	27.99%

Table 4.9: Errors and error percentages for each task morning exam 2022

Task	Errors	Percentage of students made an error on this task
Oppgave 1	51	28.88%
Oppgave 2	29	20.74%
Oppgave 3	96	35.55%

Table 4.10: Errors and error percentages for each task evening exam 2022

4.2.2 Manual Analysis of Python Code Results

Table 4.11: Results of manual analysis of Eksamen H2021

Error Description	**Count**
Incorrect variable usage	12
Incorrect dictionary usage	3
Incorrect string manipulation	4
Incorrect function usage	3
Incorrect return usage	6
TypeError	5
ValueError	3
Incorrect list manipulation	2
Incomplete code	3
Incorrect return type	1
Total	39

Table 4.12: Results of manual analysis of Eksamen S2022 Morgen

Error Type	**Count**
Incorrect variable usage in loop	1
Incorrect range for iteration	1
Incorrect usage of string/list methods	1
Naming errors (incorrect variable names, typos)	2
Incorrect logic (assertion, comparison)	1
Incorrect usage of quotes around variables	1
Incorrect index access (bounds, direction)	2
Incorrect array comparisons	1
Total	10

Table 4.13: Results of manual analysis of Eksamen S2022 Kveld

Error Type	**Count**
Naming errors (incorrect variable names, typos)	4
Incorrect usage of parentheses	1
Incorrect iteration over the function	1
Incorrect variable usage	2
Incorrect variable for list index	1
Total	9

The results of the manual error analysis conducted on Python code submitted in the exams for Eksamen H2021, Eksamen S2022 Morgen, and Eksamen S2022

Kveld are summarized in Tables 4.11, 4.12, and 4.13, respectively.

During the Eksamen H2021 (Table 4.11), a total of 39 errors were identified. It is important to note that the H2021 exam consisted of more tasks than the subsequent exams, thereby giving rise to more opportunities for errors to occur. The most common error was Incorrect Variable Usage with a count of 12, followed by Incorrect Return Usage with a count of 6, and TypeError, which occurred 5 times. Other error categories, including Incorrect Dictionary Usage, Incorrect String Manipulation, Incorrect Function Usage, ValueError, Incorrect List Manipulation, and Incomplete Code, were also present but less common, with occurrences ranging from 1 to 4.

In the Eksamen S2022 Morgen (Table 4.12), the total count of errors dropped to 10. Naming errors, specifically incorrect variable names and typos, and Incorrect Index Access (bounds, direction) became the most common errors, each with 2 occurrences. The rest of the errors, including Incorrect Variable Usage in Loop, Incorrect Range for Iteration, Incorrect Usage of String/List Methods, Incorrect Logic (assertion, comparison), Incorrect Usage of Quotes around Variables, and Incorrect Array Comparisons, were spread evenly across the remaining categories with a count of 1 each.

In the Eksamen S2022 Kveld examination (Table 4.13), the total number of errors observed was 9. Naming Errors (incorrect variable names, typos) remained the most common error, though the frequency decreased to 4 occurrences. Incorrect Variable Usage errors increased slightly in this examination to 2 occurrences. Other error types, including Incorrect Usage of Parentheses, Incorrect Iteration over the Function, and Incorrect Variable for List Index, occurred once.

4.2.3 Java Results

In a parallel study conducted by the same authors[1], an analysis of common mistakes made by novice programmers in Java was performed. The study focused on 20 manually-reviewed submissions from students at the Norwegian University of Science and Technology (NTNU). The results of the study are presented below in Table 4.14 and can serve as a comparative reference to the current Python study.

The most frequent error observed among the Java submissions was "Incompatible types" (14 occurrences), which refers to assigning a value of a different type to a variable than it was declared as. The second most common error was "Use == instead of .equals()" (5 occurrences), indicating the students' struggle with properly comparing object values. The third most common error was "No exception thrown" (5 occurrences), which occurs when a student fails to include a try-catch block in their code to handle potential exceptions.

Other common errors included "Incorrect calculation" (4 occurrences), "Incorrect return" (1 occurrence), "Try catch error" (1 occurrence), "Incorrect logic" (1 occurrence), "Undefined method" (1 occurrence), and "Incomplete code" (1 occurrence). These results reveal a pattern of challenges faced by novice Java programmers, such as type compatibility, proper use of comparison methods, and exception handling.

Error	Frequency
Incompatible types	14
Use == instead of .equals()	5
No exception thrown	5
Incorrect calculation	4
Incorrect return	1
Try catch error	1
Incorrect logic	1
Undefined method	1
Incomplete code	1

Table 4.14: Frequency of errors in Java submissions from the NTNU study

By comparing the Java results to the Python results from this study, we can gain insight into the similarities and differences in the challenges faced by novice programmers across different programming languages. This comparison may provide valuable information for educators in designing more effective programming curricula and support materials for students. The detailed implications of these findings, specifically their ramifications for educational design in programming, will be explored in the subsequent Discussion chapter, offering more thorough reflections and elaborations on this crucial aspect.

Chapter 5

Discussion

5.1 General Overview

Examining the unique errors from the home exam of Autumn 2021 and the morning and evening school exams of Autumn 2022, several notable patterns become apparent. These patterns elucidate the common challenges novice programmers encounter, and differ according to the specific exam contexts.

5.1.1 Analysis of Autumn 2021 Home Exam

In the home exam of Autumn 2021, NameError, specifically related to undefined functions such as "oppdater_matvare" and "finn_pris", was the most frequent error. Importantly, these NameErrors often occurred due to unattempted tasks, resulting in the absence of required function definitions. This trend doesn't necessarily suggest a misunderstanding of function definition or variable scoping. Rather, it could be indicative of issues related to time management, understanding task requirements, or foundational gaps in programming knowledge that deterred students from attempting these tasks. However, several similar studies on novice errors in Python identified NameError in the top two most common errors, further indicating that novice programmers tend to make these mistakes on a regular basis [25–27].

5.1.2 AssertionErrors in Exam Patterns

Several AssertionErrors, such as '0 != 99' and 'None != 0', also frequently occurred during the exam. These errors indicate that students' code ran without crashing but the output was incorrect as per the task requirements. This suggests that while students had a functional understanding of syntax to avoid run-time errors, they faced challenges in achieving the correct logic or understanding the expected outcomes of their code. This claim is supported by a similar study conducted by Veerasamy *et. al.* on an introductory Python programming e-exam [24]. They found that 69.2% of the students participating in the study made knowledge-based errors, meaning their code contained logical fallacies.

5.1.3 Analysis of Autumn 2022 Morning and Evening Exams

In the morning and evening school exams of Autumn 2022, AssertionErrors and IndexErrors were the most frequent errors. AssertionErrors such as 'False is not true' and 'None is not true' indicate that students' code ran without crashing but the output was incorrect as per the task requirements. The high frequency of IndexErrors and TypeErrors signifies difficulties with complex data structures like lists and arrays, suggesting a need for strengthening understanding and manipulation of these structures. This claim is also supported by Veerasamy *et. al.*[24], as their qualitative analysis found that "Students were not clear with index position and referencing list elements" also suggesting the same need for strengthening the students' knowledge around complex data structures.

5.1.4 Concluding Observations

In conclusion, the analysis indicates that novice programmers are grappling with challenges related to task interpretation, time management, logical reasoning, understanding of complex data structures, and achieving the correct output for given tasks. Therefore, effective pedagogical interventions should address both technical programming skills and broader problem-solving strategies. These might include a focus on understanding task requirements, managing time effectively, and cultivating an in-depth understanding of data structures, alongside strategies for debugging and validation of output.

5.2 Analysis of Error Categories Across Exams

Considering the sum of error categories from the home exam in Autumn 2021 and the morning and evening school exams in Autumn 2022, certain trends and observations become clear:

5.2.1 Autumn 2021 Home Exam

In the home exam of Autumn 2021, the two most common error categories were NameError and AssertionError. As discussed earlier, many NameErrors can be attributed to unattempted tasks, where students did not create the required functions, leading to these undefined name errors. On the other hand, the prevalence of AssertionError suggests that students' code often produced results that did not match the expected output. This implies a certain proficiency in writing code that runs without crashing, but also underscores difficulties in crafting correct logic or understanding task requirements.

5.2.2 Autumn 2022 Morning Exam

In the Autumn 2022 Morning School exam, the recurring trend of AssertionError being among the most common errors persisted, underscoring the difficulties students

encounter when correctly implementing their code logic to fulfill task requirements. However, there was a significantly lower incidence of NameError compared to the home exam. This could indicate increased task engagement or improved skills in defining functions. Another contributing factor could be the structure of the 2022 exams, which likely contained fewer tasks requiring students to utilize a function from a previous task, thus reducing the opportunity for NameErrors to occur.

5.2.3 Autumn 2022 Evening Exam

In contrast, the evening school exam of Autumn 2022 showed a considerable increase in TypeError and a decrease in both AssertionError and NameError. This shift suggests that the challenges faced by students in this context may have been more focused on correctly using and manipulating different data types, as opposed to defining correct functions or achieving correct outputs.

5.2.4 Interpreting the Data of 2022 Evening and Morning Exams

When interpreting the data from the Autumn 2022 Morning Exam and the Autumn 2022 Evening Exam, it's crucial to consider some notable elements that were observed. The structure and the scope of topics covered in both exams were nearly identical; this was because they were two versions of the same test for one class, divided solely due to the limited seating capacity in the exam venue. To avoid giving an undue advantage to the students taking the second session, the instructors took intentional measures to ensure that the exams were as closely aligned as possible, while avoiding the use of identical questions.

Moreover, efforts were made to ensure equivalent student strengths across the two sessions, with a near-proportional representation of students from various programs in each session. As such, one might anticipate similar performance across these two sessions.

Regarding the Autumn 2022 Morning Exam, the trend of AssertionError being among the most common errors persisted, further substantiating the difficulties students encounter in correctly implementing their code to meet task requirements. The NameError incidence was significantly lower compared to the home exam, potentially suggesting better task engagement or improved skills in function definition.

Conversely, the Autumn 2022 Evening Exam demonstrated a considerable increase in TypeError and a decrease in both AssertionError and NameError. This shift may imply that students' challenges in this context were more focused on correctly manipulating and using different data types, rather than defining correct functions or achieving the correct outputs.

However, it's also worth considering that despite the random selection of students from each group, the differences observed may be due to sampling effects. For instance, the increased number of TypeErrors in one of the sessions could be attributed to pure coincidence. Consequently, such factors should be considered when interpreting the results.

5.3 Error Distribution Across Tasks

Looking at the distribution of errors across different tasks in the Home Exam of 2021, and the Morning and Evening Exams of 2022, it is evident that certain tasks posed more difficulties than others.

5.3.1 Task Challenges in Autumn 2021 Home Exam

In the home exam of 2021, we observed that certain tasks exhibited a higher error rate than others. Specifically, "Oppgave 3.3" had a striking 39.88% of students making errors, with "Oppgave 3.1" close behind at 36.84%. Moreover, "Oppgave 2c" and "Oppgave 2h" both saw error rates above 30%. The reasons behind these heightened error rates can be multi-faceted, stemming from task complexity, required skill levels, or clarity of instructions.

Upon closer examination, one distinguishing feature becomes evident. Task 3, which contained the sub-tasks with the highest error rates, presents a larger, more complex problem for students to tackle. This contrasts with Task 2, which is composed of numerous smaller, independent problems. The larger context of Task 3 could introduce additional cognitive load for the students, thus resulting in more errors.

Interestingly, during the analysis, we encountered a technical issue that further complicated our understanding of the error rates. Many errors, specifically those categorized as NameErrors, were not due to mistakes in the student's code. Instead, they were the product of limitations inherent to the testing framework utilized. In python, interdependence between cells is common. A function defined in one cell may be used in another, and students were allowed to apply this intercellular logic in their solutions accross the different tasks in this exam. However, when the student's code was extracted for testing, this cell-to-cell context was lost. The Python unittest framework, which we used to test the submissions, does not support the import of functions across different files and directories. This resulted in a NameError, despite the student's code being theoretically correct within the original notebook context. This means that the errors on task 3_1 and 3_2 are inflated due to the tests recording an error message on code that is at the minimum partially correct.

Aditionally, this exam contains a large quantity of assertionerrors. This error category would likely be much larger if the presence of NameErrors was not so large. The task "Oppgave 3.1" asks students to write a function that manipulates and returns certain data. Here's an example of how the correct implementation might look:

```
def finn_pris(matvarer, let_etter):
    for vare in matvarer:
        if let_etter == vare[0]:
            return vare[1]
    return 0
```

In this task, the function finn_pris is meant to traverse a list of items matvarer, each represented as a list of an item name and its price. The function is supposed to return the price of the item specified by let_etter if it exists in the matvarer list, and return 0 if the item doesn't exist.

When students attempt this task, a common error that leads to an AssertionError is not correctly handling the case when the item is not found in the list. For example, if they forget to include the return 0 at the end of the function, the function will return None by default when the item is not found. This return value will not match the expected output of 0, resulting in an AssertionError during testing.

Another potential pitfall in this task is the incorrect comparison of the item name with the entire vare sublist (which includes the item name and the price), rather than just with the item name (vare[0]). This mistake can also lead to the function returning None instead of 0, again causing an AssertionError.

In conclusion, "Oppgave 3.1" can lead to a high number of AssertionErrors due to common oversights in handling specific cases. Proper error handling and attention to detail are crucial for passing the automated tests and successfully solving such tasks.

These findings warrant further research. The potential role of task instructions and the comparative structure of the tasks—Task 2's isolated problems versus Task 3's larger case—could be investigated in greater depth. By exploring these aspects more thoroughly, we can gain a more nuanced understanding of the challenges faced by novice programmers at NTNU.

5.3.2 Task Challenges in Autumn 2022 Morning Exam

In the morning exam of 2022, "Oppgave 2" stood out with 61.97% of students making an error, indicating a substantial difficulty faced by the majority of students. This task, potentially representing a complex problem or a difficult-to-grasp concept, may warrant a deeper exploration to identify the root cause of the common errors and address them in future teaching.

This task's main challenge seems to lie in its multi-layered conditions. It required students to correctly implement control flow statements and possess a robust understanding of Python's comparison operators.

For clarity, the task was to create a Python function that checks whether a football field is within acceptable size limits (in meters). Note that stricter size constraints apply for international matches compared to national ones.
Below is the solution to the task:

```
def ok_size(length, width, intl):
    if intl:
        return 100 <= length <= 110 and 64 <= width <= 75
    else:
        return 90 <= length <= 120 and 45 <= width <= 90
```

The function ok_size() takes three parameters: the length and width of the football field, and a boolean value indicating whether the game is international. The function should return True if the dimensions of the field are within the acceptable range based on the game type, and False otherwise.

Common pitfalls that students might encounter in this task include:

- Misunderstanding of the requirements: The task involves multiple conditions for the length and width depending on whether the game is international or not. Misunderstanding or misinterpretation of these conditions could lead to incorrect logic in the function.
- Incorrect use of comparison operators: Python's chained comparisons (e.g., 100 <= length <= 110) might be unfamiliar to some students, leading to potential errors in their implementation. However, it's worth noting that the students were not specifically required to use chained expressions. They could have also written conditions such as 'length >= 100 and length <= 110' or even split the decision across several 'if' statements instead of using just one. Therefore, it might be more precise to suggest that students may have had problems with formulating complex conditions, rather than attributing mistakes specifically to the use of chained expressions.
- Edge case handling: Students might struggle with handling edge cases. For instance, a field size that is exactly on the limit is acceptable, but one that is even slightly outside the limit is not.

As such, this task tests not only the students' understanding of control flow and comparison operators but also their attention to detail and ability to correctly interpret and implement complex requirements. Given these challenges, it's understandable that a significant proportion of students would make errors in this task. To help students overcome these challenges, future teaching could place greater emphasis on conditional logic, operator usage, and careful reading of problem statements. The most common error AssertionError('False is not true : False') occurs on this task. This error is raised when an assert statement fails. In the context of unittest, assert statements are used to verify that the output of a function is as expected. If the output does not match the expected result, an AssertionError is raised.

In relation to the ok_size() function task, this error would suggest that a test case expected the function to return True, but the function instead returned False. The details of why this happened would depend on the specific test case and the student's implementation of the ok_size() function.

One potential cause for this error that many students seem to have encountered could be that the student misunderstood or incorrectly implemented the requirements for acceptable field sizes, particularly for edge cases. For example, they might have used < or > instead of <= or >=, which would result in their function incorrectly returning False for a field size that is exactly on the limit. Such edge cases seem to be a common source of errors in this task. This aligns with the findings of Robins *et. al.*[45] who discussed common difficulties students face in learning to program, including misunderstanding requirements and trouble handling edge

cases. Furthermore, a multinational, multi-institutional study by McCracken *et. al.*[46] found that students often struggle with accurately implementing program requirements, especially in terms of edge cases and boundary conditions.

5.3.3 Task Challenges in Autumn 2022 Evening Exam

In contrast, the evening exam of 2022 presented the most significant challenge in "Oppgave 3", with a 35.55% of students making errors. This highlights "Oppgave 3" as an area where students have a higher rate of errors in comparison to the other tasks. Oppgave 3 on both the morning and evening exams are very similar in format, and logic, and also seem to be quite close in student errors. This is because the tasks are designed to test for identical knowledge. In "Oppgave 3", students were asked to program the function "sum_near_whole(A)" which receives a two-dimensional numpy array as a parameter and is expected to return the sum of only certain numbers within the array. Specifically, the function should consider only the numbers that are located above, below, or beside a whole number. This task essentially tests the students' understanding of multi-dimensional array manipulation using the numpy library, as well as their ability to apply conditional logic to identify and sum only the numbers that fulfill the given condition. The complexity of the task is further heightened due to the need to consider the position of numbers relative to whole numbers within the array. Consequently, errors in this task could emerge from multiple sources such as difficulties in correctly identifying and accessing elements adjacent to whole numbers in the array, or confusion in distinguishing between whole numbers and numbers with decimal parts. This task underscores the importance of a strong understanding of array manipulation and conditional logic in Python programming, and serves as a challenging exercise for novice programmers.

5.4 Identifying Common Error Types

From the manual analysis of the exams, it is possible to identify some common error types that students committed. This analysis further emphasizes the necessity of reinforcing certain coding concepts and practices in teaching. Here is a summary of the main findings:

5.4.1 Errors in Autumn 2021 Exam (Eksamen H2021)

In the 2021 exam, the most common error was related to incorrect variable usage (12 occurrences), followed by incorrect return usage (6 occurrences) and TypeErrors (5 occurrences). Other common errors include incorrect string manipulation, dictionary usage, function usage, and list manipulation. Furthermore, some students produced incomplete code or returned an incorrect type from a function. This suggests that students may have struggled with understanding the specific requirements of tasks or had difficulty managing different data types in Python.

5.4.2 Errors in Autumn 2022 Morning Exam (Eksamen S2022 Morgen)

For the 2022 morning exam, there was a wider distribution of error types, with naming errors (including incorrect variable names and typos) and incorrect index access being the most frequent. Students also struggled with the incorrect usage of string/list methods, variable usage in loops, and logical assertions or comparisons. It seems that a more detailed understanding of Python's built-in methods and correct indexing is necessary, along with a stronger grasp of logical constructs in Python. Also, the high occurrence of naming errors highlights the importance of proper code writing and debugging practices.

5.4.3 Errors in Autumn 2022 Evening Exam (Eksamen S2022 Kveld)

In the 2022 evening exam, the most common error was again related to naming, followed by incorrect variable usage. Other errors include incorrect usage of loops, conditions, and return usage. These patterns suggest the need for reinforcement of fundamental coding concepts, including the appropriate use of variables, loops, and conditionals, as well as an understanding of function behavior in Python, including how to correctly return values from a function.

5.5 Commonalities and Differences in Errors Across Home and School Exams

Across the home and school exams, certain errors emerged as common stumbling blocks for novice programmers. However, there were also significant differences that reflect the unique challenges posed by different exam contexts and programming languages.

5.5.1 Common Errors

One of the most consistent error types across both the home and school exams was "Incorrect variable usage". This type of error occurred when students did not utilize variables correctly according to the task requirements or the conventions of the programming language. This suggests that understanding and manipulating variables is a fundamental skill that students across different programming languages and contexts struggle with.

Another shared error was "Incorrect function usage" or issues related to function definition, which featured in both the home and morning school exams. This points to potential difficulties in comprehending task requirements or possibly gaps in foundational programming knowledge related to defining and using functions effectively.

Both the home exam and the school exams saw instances of "Incomplete code", where students failed to fully complete the tasks at hand. This type of error could

be indicative of challenges related to time management, task interpretation, or perceived difficulty of the task.

5.5.2 Differences in Errors

Despite these commonalities, there were also significant differences in error types across the exams, likely reflective of the unique challenges of different exam contexts and the specific characteristics of the programming languages used.
One notable distinction between the exams was the occurrence of the "NameError", often resulting from unattempted tasks leading to missing function definitions, in the home exam. This error was notably absent from the school exams. This could possibly be attributed to differences in the design of the exams. Specifically, the home exam might have had more interconnected subtasks where students were supposed to use a function defined in a previous task, leading to more instances of "NameError" when those prior tasks were not imported correctly to the testing environment. On the other hand, the school exams from the following year might have had fewer such interconnected tasks, thus reducing the likelihood of this error. Alternatively, the absence of "NameError" in the school exams could also suggest improvements in task engagement or function definition skills over time, or it could reflect changes in the complexity or framing of the tasks between the exams.
In the school exams, "AssertionError" was a common error, implying that while students could write code that runs without crashing, they often failed to implement the correct logic to meet task requirements. This error was less prominent in the home exam, suggesting a possible shift in challenges faced by students over time or under different exam conditions.

The school exams also saw a higher frequency of "IndexError" and "TypeError", suggesting struggles with complex data structures and their manipulation – a theme that was less prevalent in the home exam.

An important aspect to consider when interpreting these results is the discrepancy between the available tools and environments in the home and school exams. At home, students had access to a range of resources that were not available to them in the school exam setting. Specifically, they could freely use their preferred programming environment, enabling them to write and test their code more comfortably. Additionally, they could search the internet for solutions to similar problems, which might help in understanding and fixing errors they encounter. This access to tools and resources likely facilitated a more iterative and dynamic problem-solving process, where students could validate their code and adjust it based on feedback and additional information.
On the other hand, during the school exams, students were confined to a more restrictive environment. They were unable to test their code or seek online help, which might have led to a higher occurrence of certain types of errors. This difference in the available resources and environments could contribute to the variations in error types observed between the home and school exams. Hence, it's crucial to

consider these factors when interpreting the study's findings and extrapolating them to other contexts.
In addition to these factors, it is necessary to address the potential issue of academic dishonesty during home exams. According to a study by Sindre *et. al.*[47], there is evidence that some students might have cheated during these exams. The paper analyses the same home exam and found several cases f cheating. However, it's essential to note that while some instances of cheating were detected, others might have slipped through unnoticed. Detected cases often involved students receiving code from others on specific tasks that had several variants. Nonetheless, other forms of dishonesty, such as having a more proficient individual complete the exam on behalf of the student, would likely have evaded this detection method. This potential for undetected dishonesty poses a significant challenge in evaluating students' actual programming abilities based on home exam performance.
Another consideration worth discussing involves the classification of "NameError" in our study. We observed that "NameError" often resulted from blank answers, which aren't necessarily indicative of a name error from the student's perspective, but rather a byproduct of our testing framework. In these instances, the absence of function definitions due to unattempted tasks was flagged as "NameError". However, it's important to note that such cases fundamentally differ from instances where students have genuinely committed a "NameError".
For instance, true "NameError" occurrences might arise from mistakes such as mistyping a variable name or attempting to use a variable before it's defined. These errors reflect a misunderstanding or oversight in the student's code and can provide meaningful insights into the common pitfalls encountered by novice programmers. In contrast, a "NameError" resulting from a blank answer does not offer the same level of insight into a student's understanding or proficiency in Python programming.
Therefore, future analyses could benefit from distinguishing between these two types of "NameError" occurrences. This differentiation would not only provide a more nuanced understanding of the errors but also contribute to a more accurate representation of students' programming proficiency and the challenges they face.

5.5.3 Implications

The comprehensive analysis of common and distinct errors across different exam settings provides a valuable foundation to inform pedagogical practices for teaching programming. Recognizing the challenges novice programmers face is critical for adapting teaching strategies that effectively address these issues.

Our findings suggest that the understanding and manipulation of variables and functions pose consistent challenges across all exam contexts. Educators should emphasize these foundational concepts to reinforce students' programming skills. Similarly, the prevalent difficulties with data structures and edge case scenarios in the school exams indicate a need for additional practice in these areas.

Moreover, the differences in error types between home and school exam contexts

underscore the need for adaptive teaching strategies. These should cater to diverse learning environments and take into account the specific characteristics and requirements of different programming languages.

However, the suggestions derived from our study should not be considered in isolation. It is crucial to compare our findings and consequent advice with similar literature. For example, other studies might identify similar issues, leading to parallel or complementary recommendations for pedagogical approaches. These existing insights can provide a broader context for our findings and enable a more informed decision-making process for educators. Unfortunately, it is difficult to compare the results from our manual analysis to similar existing studies as their error categorization are less detailed compared to our own. Thus, further research on common novice errors in Python, using manual analysis with clearly defined error categories, could provide valuable insight and confirm the accuracy of our analysis.

In conclusion, the amalgamation of findings from our study and similar literature can lead to a comprehensive set of practical recommendations for educators. These can then guide the design and implementation of more effective teaching strategies in programming education, tailored to address specific learning challenges encountered by novice programmers.

5.6 Summary

To summarize, it appears that novice programmers often face challenges in understanding and implementing task requirements, managing different data types in Python, and utilizing proper code writing and debugging practices. In order to improve student performance in these areas, teaching strategies could involve focused training on understanding task requirements, handling different data types, implementing logic correctly, and cultivating best practices in code writing and debugging.

Chapter 6

Conclusion

6.1 General Findings

The study provided an in-depth analysis of the common errors made by novice programmers in Java and Python at the Norwegian University of Science and Technology. Through a systematic examination of exam submissions, we gained valuable insights into the challenges faced by these students. The results demonstrated that the most frequent errors made were not necessarily language-specific but were rather linked to fundamental programming concepts.

The findings underscored a noteworthy trend: many of the most frequently encountered errors weren't specifically tied to the particularities of Java or Python. Instead, they seemed to revolve around core concepts inherent to programming as a discipline. Errors such as 'AssertionError', 'TypeError', 'IndexError' and 'NameError' were common among the data sets. This signifies fundamental misunderstandings related to variable declaration, function usage, and conditional logic. Furthermore, errors related to data structures like lists and arrays were also notably common. This argument is opposed by the findings from our literature review, claiming that common errors are highly language dependant. This is likely a result of differences between the methodology of different studies, such as error categorization and data gathering/analysis. Existing studies mostly looked into compilation and run-time errors, which is language dependant causing a comparison between studies on different languages to come to the same conclusion.

Interestingly, there was a higher prevalence of type-related errors in Java submissions. This might be attributed to Java's statically-typed nature, which enforces stricter type checking rules during compile-time, compared to Python's dynamic typing. Misunderstandings or negligence regarding data type compatibility can lead to such errors, underscoring the importance of a solid understanding of data types and their rules in Java.

However, it's important to note that the above comparison between Java and Python is made with caution due to differences in the nature of exams and the small sample size of Java. Although certain patterns and trends are evident, these findings should be validated with larger and more homogenous data sets to draw

more definitive conclusions about the distinct challenges faced by students of Java and Python.

Understanding these general findings is pivotal for educators and curriculum designers. It encourages a focus on reinforcing understanding of fundamental programming concepts, which could lead to marked improvements in learning outcomes for novice programmers. The results underline the significance of a robust foundational knowledge in programming education, potentially influencing future curriculum design and teaching methodologies.

6.2 Limitations

When interpreting the findings of this study, several limitations must be acknowledged.

First and foremost, the unit testing script used in this research was specifically designed for Python code. As a result, the transferability of these results to other programming languages may be limited, potentially constraining the broader applicability of our findings within the extensive domain of computer programming education.

Furthermore, our script was designed to focus on a specific set of programming concepts. This focus may not encapsulate the entirety of skills and knowledge required in more advanced programming courses or in professional scenarios. Hence, our conclusions might not fully reflect students' programming expertise or their ability to apply these skills in diverse or complex situations.

Another important consideration is the specificity of our participant group and the details of the exams used in this study. Certain demographic attributes, levels of experience, or instructional methodologies may have influenced the outcomes, and these factors might not perfectly represent the larger population of programming students. This specificity could affect the generalizability of our findings, a factor to be accounted for when applying these conclusions to varied situations or learner populations.

A noteworthy limitation of our study is its emphasis on procedural programming in Python, largely overlooking object-oriented programming. While students did use object method calls for elements such as lists, strings, and numpy arrays, the course did not explicitly focus on enhancing comprehension and application of object-oriented programming (OOP). As such, students were not expected to define their own classes or solve problems using an object-oriented approach.

This emphasis on procedural programming suggests that our findings might not represent the spectrum of possible errors or misunderstandings related to object-oriented concepts. The limited focus on object-oriented programming means our study may not fully capture students' competence or deficiencies in this critical area. Given the importance of OOP in many settings, this omission limits the breadth of our study's conclusions and their applicability in environments where OOP is a key element.

To improve the comprehensiveness and utility of future research, studies should strive to incorporate a broader range of both procedural and object-oriented programming concepts. This approach would provide a more holistic view of student capabilities

and the potential pitfalls in different programming paradigms, thereby enhancing the relevance and value of the research findings across a wider range of educational and professional contexts.

6.3 Future Research

This analysis provides valuable insights into the types of errors encountered by novice programmers in different exam contexts. However, it also sets the stage for further investigation and poses several questions that warrant deeper exploration.

Understanding Root Causes of Errors: Further research could delve into the root causes of common errors such as NameErrors and AssertionErrors. Our observations revealed that the NameErrors often stemmed from incomplete code, or code that attempts to execute undefined functions, and that AssertionErrors executed without overt errors, but failed to yield the correct results. Identifying whether these errors are primarily due to lack of knowledge, misunderstandings about task requirements, issues related to time management, or limited access to resources could shed light on the underlying factors. This deeper understanding could enable the development of more effective teaching interventions and strategies.

Examining the Impact of Teaching Strategies: There is potential to explore the effectiveness of various teaching strategies on error reduction. It is pertinent to understand which methods are most beneficial in helping students comprehend and rectify their errors. Do specific interventions work better for certain types of errors, or are they more effective under different exam conditions? Answering these questions could significantly enhance the teaching and learning experience in programming education.

Exploring Other Programming Languages: While this study is focused on Python code, future research could broaden its scope to include other programming languages. Comparing error patterns across a diverse range of languages could enhance our understanding of how students learn, their common stumbling blocks, and how these challenges may vary with different programming languages.

6.3.1 Potential Research Directions

Future research could delve deeper into the impact of various pedagogical strategies on the types of errors made by novice programmers. Investigating these connections could provide insights into how teaching methodologies influence learning outcomes in programming education. Some potential areas of exploration could include:

Peer Programming: Peer programming, also known as pair programming, involves two students working together on the same programming task. This collaborative approach could potentially reduce errors, as one student codes while the other reviews the work in real time, allowing immediate feedback and error correction [48]. Future research could examine how this cooperative method influences the frequency and types of errors made by novice programmers.

Interactive Online Platforms: Interactive online platforms provide an engaging way for students to learn programming concepts. They often include real-time feedback and opportunities to practice coding skills in a variety of contexts. Research such as that done by Nariman[49] could explore how the use of these platforms impacts the types of errors students make and how quickly they can correct these errors.

Flipped Classroom Models: In a flipped classroom model, students review lecture materials at home and use class time for problem-solving and practical application of the concepts learned. This model allows for more personalized attention from the instructor and a deeper engagement with the material. Future studies, like the one conducted by Amresh *et. al.*[50], could investigate the impact of this teaching model on the types of errors made by novice programmers and their overall proficiency in programming. The aforementioned study finds that the learning method shows promise, but the research also highlights that some students find the new approach to be intimidating and overwhelming at times.

By examining these and other pedagogical strategies, future research could provide valuable insights into how best to support novice programmers. This could contribute to the design of more effective teaching methods and resources, thereby improving learning outcomes in programming education.

Exploring Object-Oriented Programming (OOP): A critical aspect of programming that warrants further research is the exploration and comparison of errors within Object-Oriented Programming (OOP). Given the widespread use of OOP in many modern programming languages, it is important to understand the unique challenges novice programmers might face while dealing with classes, objects, inheritance, and other OOP principles. Future studies could compare the error patterns of procedural and object-oriented programming. This could not only shed light on the nuances of mistakes made in different programming paradigms but also contribute to developing pedagogical strategies that address specific areas of difficulty in OOP. This in-depth exploration of OOP could help bridge the gap in the existing literature and enrich our understanding of programming education as a whole.

6.4 Connection with Research Questions

Our research embarked on a journey to explore and answer four specific questions:

1. Which common errors are made by novice programmers in Java?
2. Which common errors are made by novice programmers in Python?
3. Are there any correlations between common errors made by novice programmers in Java and Python?
4. Are there differences in the errors made during home exams and school exams?

Through a systematic analysis of exam submissions from an introductory programming course at NTNU, we managed to identify the common errors made by novice programmers in both Java and Python (Q1 and Q2). We found that errors such as

'AssertionError', 'TypeError', 'IndexError', and 'NameError' were prevalent across both languages. This revealed that students often struggled with the same core programming concepts, regardless of the language being used.

Moreover, our research discerned a correlation between the errors made by students in Java and Python (Q3). Many of the mistakes were not exclusive to a particular language, but were rooted in fundamental misunderstandings of essential programming principles. This highlighted the universality of certain challenges faced by novice programmers, regardless of the programming language in use.

However, the differences in errors made during home exams and school exams (Q4) require further investigation. Our study did not gather sufficient evidence to conclusively determine if the setting significantly influenced the type or frequency of errors made by students. The potential impact of the exam context on students' programming mistakes remains a ripe area for future research.

6.5 Revisiting the Introduction

In the beginning, we set out to explore the errors that novice programmers typically make in Java and Python, within the specific context of the Norwegian University of Science and Technology. The goal was to shed light on the struggles faced by beginner students in their initial encounter with programming. By comprehending these challenges, we hoped to inform pedagogical strategies that could potentially alleviate these hurdles and enhance learning outcomes.

Today, we conclude our journey having made significant strides towards achieving our initial goals. We not only identified the common errors made by novice programmers in both Java and Python, but we also unveiled the foundational nature of these errors, irrespective of the programming language. We found that the struggles of these students often stemmed from the basic principles of programming rather than the specific syntax or semantics of Java or Python.

However, this conclusion is not the end; instead, it signifies a new beginning. It reveals a landscape ripe for further investigation and raises several questions for future research. We hope that our study contributes to the growing body of knowledge in the field of computer science education and serves as a stepping stone for future investigations exploring programming learning and teaching strategies.

Bibliography

[1] O.-C. Bjerkeset, S. W. Cook and G. Sindre, 'Common mistakes made by novice programmers in java,' *IT3915 - Specialization project for Computer Science*, 2023.

[2] M. Kölling and D. Barnes, *Objects First with Java: A Practical Introduction Using BlueJ*. Pearson, 2016, ISBN: 978-0134477367.

[3] M. Kölling and J. Rosenberg, 'Tools and techniques for teaching objects first in a java course,' 1999.

[4] M. C. Jadud, 'Methods and tools for exploring novice compilation behaviour,' 2006.

[5] E. S. Tabanao, M. M. T. Rodrigo and M. C. Jadud, 'Identifying at-risk novice java programmers through the analysis of online protocols,' 2009.

[6] E. S. Tabanao, M. M. T. Rodrigo and M. C. Jadud, 'Predicting at-risk novice java programmers through the analysis of online protocols,' *Proceedings of the Seventh International Workshop on Computing Education*, 2011.

[7] M. M. T. Rodrigo, E. Tabanao, M. B. E. Lahoz and M. C. Jadud, 'Analyzing online protocols to characterize novice java programmers,' *Philippine Journal of Science*, 2009.

[8] J. T. C. Mow, 'Analyses of student programming errors in java programming courses,' *Journal of Computing*, 2012.

[9] N. C. C. Brown, M. Kölling, D. McCall and I. Utting, 'Blackbox: A large scale repository of novice programmers' activity,' *SIGCSE '14: Proceedings of the 45th ACM technical symposium on Computer science education*, 2014.

[10] D. McCall and M. Kölling, 'A new look at novice programmer errors,' *Transactions of Computing Education*, 2019.

[11] D. McCall and M. Kölling, 'Meaningful categorisation of novice programmer errors,' *Institute of Electrical and Electronics Engineers*, 2014.

[12] M. Hristova, A. Misra, M. Rutter and R. Mercuri, 'Identifying and correcting java programming errors for introductory computer science students,' *ACM SIGCSE Bulletin*, 2003.

[13] S. Garner, P. Haden and A. Robins, 'My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems,' *ACE '05: Proceedings of the 7th Australasian conference on Computing education*, 2005.

[14] K. Goldman, P. Gross, C. Heeren and G. Herman, 'Identifying important and difficult concepts in introductory computing courses using a delphi process,' *ACM SIGCSE Bulletin*, 2008.

[15] L. C. Kaczmarczyk, E. R. Petrick, J. P. East and G. L. Herman, 'Identifying student misconceptions of programming,' *Proceedings of the 41st ACM technical symposium on Computer science education*, 2010.

[16] R. Caceffo, P. Frank-Bolton, R. Souza and R. Azevedo, 'Identifying and validating java misconceptions toward a cs1 concept inventory,' *the 2019 ACM Conference*, 2019.

[17] T. Flowers, C. A. Carver and J. Jackson, 'Empowering students and building confidence in novice programmers through gauntlet,' *34th Annual Frontiers in Education, 2004. FIE 2004.*, 2004.

[18] J. Jackson, M. Cobb and C. Carver, 'Identifying top java errors for novice programmers,' *Institute of Electrical and Electronics Engineers*, 2006.

[19] A. Altadmri and N. C. C. Brown, 'Investigating novice programming mistakes: Educator beliefs vs. student data,' *Proceedings of the tenth annual conference on International computing education research*, 2014.

[20] A. Altadmri and N. C. C. Brown, '37 million compilations: Investigating novice programming mistakes in large-scale student data,' *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015.

[21] N. C. C. Brown and A. Altadmri, 'Novice java programming mistakes: Large-scale data vs. educator beliefs,' *Association for Computing Machinery (ACM)*, 2017.

[22] J. Bulmer, A. Pinchbeck and B. Hui, 'Visualizing code patterns in novice programmers,' *the 23rd Western Canadian Conference*, 2018.

[23] P. O. jegede, E. Olajubu, A. O. Ejidokun and I. Elesemoyo, 'Concept–based analysis of java programming errors among low, average and high achieving novice programmers,' *Journal of Information Technology Education: Innovations in Practice*, 2019.

[24] A. K. Veerasamy, D. D'Souza, M.-J. Laakso and G. Herman, 'Identifying novice student programming misconceptions and errors from summative assessments,' *Journal of Educational Technology Systems*, 2016.

[25] R. Smith and S. Rixner, 'The error landscape: Characterizing the mistakes of novice programmers,' *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019.

[26] T. Kohn, 'The error behind the message: Finding the cause of error messages in python,' *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019.

[27] B. Jeffries, J. A. Lee and I. Koprinska, '115 ways not to say hello, world!: Syntax errors observed in a large-scale online cs0 python course,' *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education*, 2022.

[28] F. Johnson, S. McQuistin and J. O'Donnel, 'Analysis of student misconceptions using python as an introductory programming language,' *Proceedings of the 4th Conference on Computing Education Practice 2020*, 2020.

[29] A. Ettles, A. Luxton-Reilly and P. Denny, 'Common logic errors made by novice programmers,' *the 20th Australasian Computing Education Conference*, 2018.

[30] D. M. Blei and M. I. Jordan, 'Variational inference for dirichlet process mixtures,' *Bayesian Analysis.*, 2006.

[31] A. Emerson, A. Smith, F. J. Rodriguez, E. N. Wiebe, B. W. Mott, K. E. Bover and J. C. Lester, 'Cluster-based analysis of novice coding misconceptions in block-based programming,' *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 2020.

[32] M. J. Johansen, 'Errors and misunderstandings among novice programmers,' *UiO, Unniversity of Oslo*, 2015.

[33] K. M. Rørnes, R. K. Runde and S. M. Jensen, 'Students' mental models of references in python,' 2019.

[34] *Inspera*, `https://www.inspera.com/`, Accessed: 2022-12-05.

[35] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2014.

[36] R. B. Johnson and A. J. Onwuegbuzie, 'Mixed methods research: A research paradigm whose time has come,' *Educational researcher*, vol. 33, no. 7, pp. 14–26, 2004.

[37] A. Tashakkori and C. Teddlie, *Sage handbook of mixed methods in social & behavioral research*. Sage Publications, 2010.

[38] V. L. Plano Clark and N. V. Ivankova, *Mixed methods research: A guide to the field*. Sage Publications, 2016.

[39] J. Creswell and V. Plano Clark, 'Designing and conducting mixed methods research,' 2017.

[40] R. K. Yin, *Case Study Research: Design and Methods*. Sage Publications, 2003.

[41] C. Teddlie and A. Tashakkori, *Foundations of mixed methods research: Integrating quantitative and qualitative approaches in the social and behavioral sciences*. Sage, 2009.

[42] B. Flyvbjerg, 'Five misunderstandings about case-study research,' *Qualitative Inquiry*, vol. 12, no. 2, pp. 219–245, 2006.

[43] H. Snyder, 'Literature review as a research methodology: An overview and guidelines,' *Journal of Business Research*, 2019.

[44] B. Kitchenham and S. M. Charters, 'Guidelines for performing systematic literature reviews in software engineering,' 2007.

[45] A. Robins, J. Rountree and N. Rountree, 'Learning and teaching programming: A review and discussion,' *Computer science education*, vol. 13, no. 2, pp. 137–172, 2003.

[46] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas and I. Utting, 'A multi-national, multi-institutional study of assessment of programming skills of first-year cs students,' *ACM Sigcse Bulletin*, vol. 33, no. 4, pp. 125–180, 2001.

[47] G. Sindre and B. Haugset, 'Techniques for detecting and deterring cheating in home exams in programming,' 2022. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3049933.

[48] J. C. Carver and L. Jaccheri, 'Exploring the potential of peer programming in undergraduate computer science education,' *ACM SIGCSE Bulletin*, 2006.

[49] D. Nariman, 'Impact of the Interactive e-Learning Instructions on Effectiveness of a Programming Course,' *Complex, Intelligent and Software Intensive Systems*, vol. 1194, p. 588, 2021. DOI: 10.1007/978-3-030-50454-0_61.

[50] A. Amresh, A. R. Carberry and J. Femiani, 'Evaluating the effectiveness of flipped classrooms for teaching CS1,' *Proceedings - Frontiers in Education Conference*, pp. 733–735, Oct. 2013, ISSN: 0190-5848. DOI: 10.1109/FIE.2013.6684923.

Appendix A

Exam Tasks and Suggested Solutions

Figure A.1: Exam Tasks and Suggested Solutions Home Exam 2021

Tasks and Suggested Solution for Home Exam 2021

Oppgave 2 (40%)

Oppgave 2 består av 9 ulike oppgaver (2a - 2i) med programmering og dra-og-slipp / innfylling i programkode. Til sammen utgjør disse oppgavene 40% av karakteren. På oppgaver hvor du kan teste om koden din virker, vil du normalt score mer poeng ved å ha et program som passerer iallfall noen av testene, enn ved å ha et program som prøver å få til alt, men får kjørefeil og ikke passerer noen av testene. Det er likevel bedre å levere noe kode selv om den ikke virker, enn å levere blankt, da sensor med et levert svar kan ha mulighet til å se over koden manuelt og vurdere om det kan gis litt poeng for delvis riktig tankegang.

2a - Max Manus (3%)

Pythons standardfunksjon max() returnerer den største av verdiene i en sekvens. For eksempel vil max("manus") gi verdien "u" fordi u er lengst ut i alfabetet av bokstavene i manus. Tilsvarende vil min("manus") gi verdien "a". Skriv en funksjon alfa(streng, tall) som får inn en tekststreng pluss et tall. Hvis tallet er 0 skal funksjonen returnere det minste tegnet i strengen (dvs. fremst i alfabetet), hvis tallet er noe annet enn 0, skal den returnere det største tegnet i strengen.

```
def alfa(streng, tall):
    if tall == 0:
        return min(streng)
    else:
        return max(streng)
```

2b - Byer i Belgia (3%)

Gitt ei ordbok kalt geografi som har land som nøkkelverdi, og for hver nøkkel ei liste av byer i dette landet. Et eksempel på mulig innhold er: geografi = {'Irland': ['Dublin', 'Cork'], 'Polen': ['Lodz', 'Krakow', 'Gdansk'], 'Belgia': ['Bryssel', 'Gent', 'Liege', 'Namur'] } Funksjonen ant_land(geografi) skal returnere antallet land som fins i ordboka. Med eksemplet over skal dette kallet returnere tallet 3 fordi det fins info om tre land i ordboka. Hint: funksjonen len() gir antall element i f.eks. strenger, lister, tupler, mengder, ordbøker. NB: Funksjonen din skal også virke for ordbøker med annet antall land enn eksemplet gitt her, men strukturen kan alltid antas å være den samme (land som nøkkelverdi, så liste med byer for hvert land).

```
def ant_land(geografi):
    return len(geografi)
```

2c - Sjekk tall (4%)

Skriv en funksjon sjekk(tall) som får inn et tall. Hvis tallet er et oddetall, skal funksjonen returnere strengen "odde", hvis partall returnere strengen "par", og hvis tallet har en desimaldel > 0, skal funksjonen returnere strengen "des". (Dvs. f.eks., 3 -> "odde", 3.0 -> "odde", 3.1 -> "des", 4.0 -> "par", 4.2 -> "des") NB: Funksjonen skal returnere tekststrengen, IKKE printe den.

```
def sjekk(tall):
    if tall % 2 == 0:
```

Figure A.2: Exam Tasks and Suggested Solutions Evening School Exam 2022

Tasks and Solution from Evening School Exam 2022

Oppgave 1

(a) sum_larger (6%)

Funksjonen **sum_larger(numlist, n)** skal få inn ei liste med heltall (numlist) og et heltall n. Den skal returnere summen av de tallene i lista **som er større enn n**.
Eksempel på kjøring:
>>> sum_larger([2,5,4,7,3,8], 5)
15
Dette kallet returnerer 15 fordi bare tallene 7 og 8 (som er større enn 5) kommer med i summen, altså 7+8.
Skriv koden for funksjonen sum_larger()

```
def sum_larger(numlist, n):
    result = 0
    for element in numlist:
        if element > n:
            result += element
    return result

def sum_larger(numlist, n):
    return sum(x for x in numlist if x > n)

import numpy as np
def sum_larger(numlist, n):
    A = np.array(numlist)
    return np.sum(A[A>5])
```

Å gå i løkke gjennom ei liste gjøre en eller annen aggregeringsoperasjon – f.eks. summere tallene – er nærmest et standardproblem i intro prog.fag. Her er det lagt inn en ekstra liten komplikasjon med at man ikke skal summere alle tallene, ellers ville det vært for lett (bare putte hele lista inn i den innebygde sum-funksjonen)

Her viser vi tre eksempler på løsning, først det typiske skoleeksemplet med å initialisere variabel til null, gå i løkke gjennom, teste betingelsen, summere opp, og returnere etter løkka (NB: viktig at return er rykket ut på rett marg, retur inni løkka vil gjøre at vi avslutter etter det første tallet). Mange har nok løsninger cirka som denne, det er selvsagt også helt greit å bruke for-løkke som itererer på indeks, eller while-løkke.

Det andre eksemplet er en mer kompakt løsning, putter et generator-objekt inn i Python sin sum-funksjon. Neppe mange som har gjort akkurat dette, men en del kan ha gjort nesten det samme med *list comprehension*, dvs.

sum([x for x in numlist if x > n]) , selvsagt også helt ok, selv om man da bruker litt mer minne på å lage ekstra liste. Det fins også andre fikse muligheter man kan bruke, f.eks. Python sin filter()-funksjon, men den var ikke i pensum, så vil nok være få eller ingen løsninger med den (og er vel dessuten overkill for å så enkelt problem som dette)

Nederst, konverterer lista til et numpy array – siden det ofte gir mulighet for kompakte løsninger på problemer som dette. For eksempel vil notasjonen i nederste kodelinje gjøre at vi får et array med bare tallene > 5, som vi så kan putte inn i np.sum-funksjonen.

Figure A.3: Exam Tasks and Suggested Solutions Morning School Exam 2022

Tasks and Suggested Solutions to Morning School Exam 2022

Oppgave 1

6 sum_except (6%)

Funksjonen **sum_except(numlist, n)** skal få inn ei liste med heltall (numlist) og et heltall n. Den skal returnere summen av tallene i lista, **unntatt** eventuelle forekomster av tallet n.

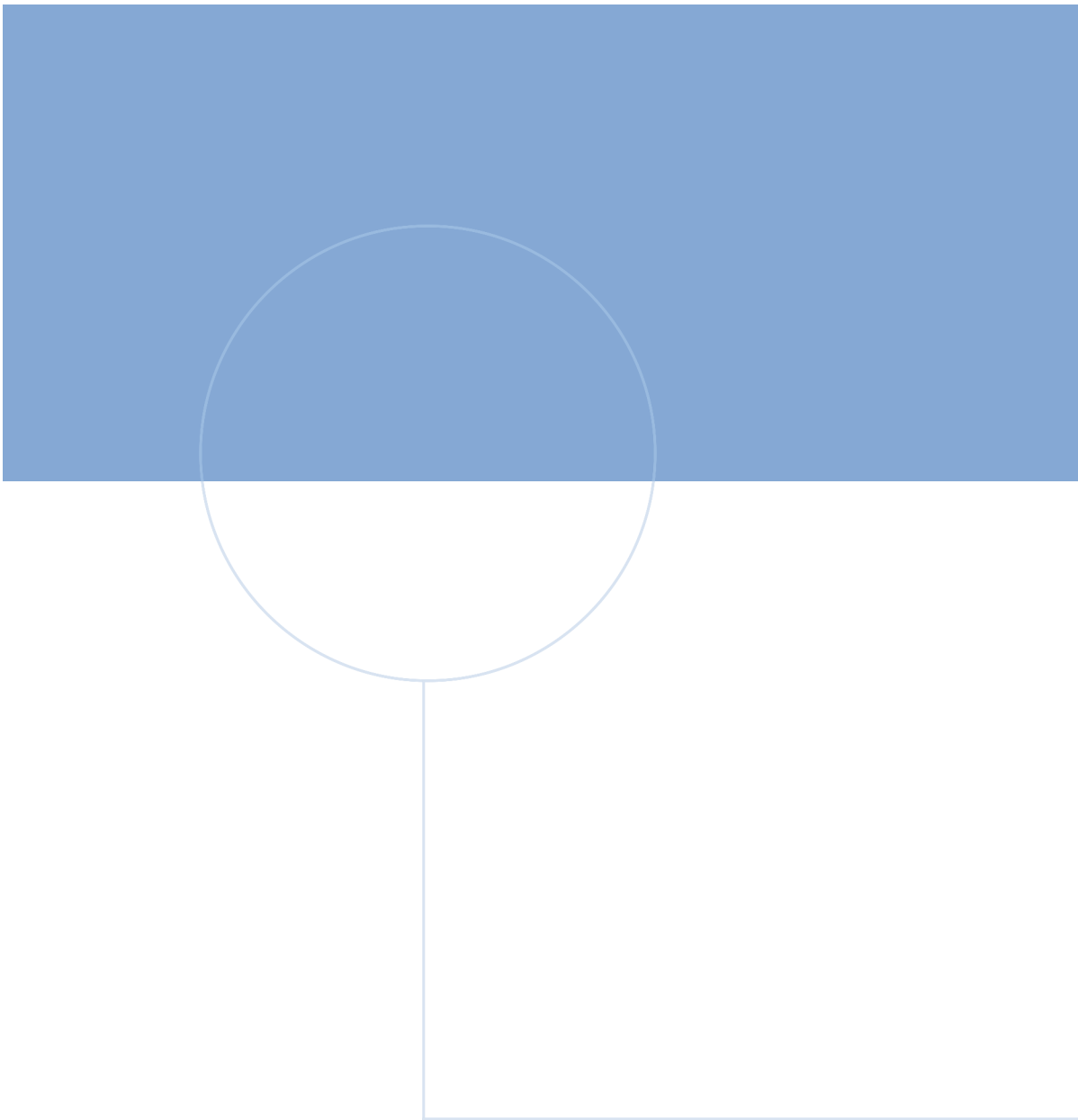
Eksempel på kjøring:
>>> sum_except([3, 4, 3, 7], 3)
11
Dette kallet returnerer 11 fordi de to 3-tallene i lista ikke blir med i summen, slik at resultatet blir 4+7.
Skriv koden for funksjonen sum_except()

```
def sum_except(numlist, n):
    result = 0
    for element in numlist:
        if element != n:
            result += element
    return result

def sum_except(numlist, n):
    return sum(x for x in numlist if x != n)

import numpy as np
def sum_except(numlist, n):
    A = np.array(numlist)
    return np.sum(A[A!=n])
```

Selv om dette er – og var ment å være - en forholdsvis enkel oppgave, kan den gjøres på mange ulike måter, bildet opp til høyre viser 3 alternativ.

- Øverst: rett fram løsning med for-løkke og if-setning. For-løkka kunne alternativt også ha iterert på indeks, eller man kunne ha brukt while-løkke (selv om litt mer tungvint).
- Midt: mer kompakt løsning, putter et generator-objekt inn i Python sin sum-funksjon. Neppe mange som har gjort akkurat dette, men en del kan ha gjort nesten det samme med *list comprehension*, dvs. **sum([x for…])**, selvsagt også helt ok, selv om man strengt tatt ikke trenger bruke minneplass på å lage ei ekstra liste her.
- Nederst: løsning hvor man konverterer lista til et numpy array og benytter seg av en kompakt notasjon som numpy har for da å lage tilsvarende array uten tallet n, som puttes inn i numpy.sum-funksjonen.

Man kan bruke hvilken fremgangsmåte man vil – en av de tre over eller noe annet. Perfekt fungerende løsning vil få full pott. Ikke-perfekte løsninger vurderes etter hvor stor andel man har greid av det som funksjonen skulle gjøre.



NTNU

Norwegian University of
Science and Technology