The Error that is the Error message

Comparing information expectations of novice programmers against the information in Python error messages.

Bart Heemskerk

The Error that is the Error message

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bart Heemskerk
born in Leiden, the Netherlands

TUDelft

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2020 Bart Heemskerk.

The Error that is the Error message

Author: Bart Heemskerk
Student id: 4143469
Email: B.J.Heemskerk@student.tudelft.nl

Abstract

Learning to program is not a easy task, as has become evident from the abundance of research papers concerning the subject. One of the learning barriers of learning a new programming language is understanding their error message, as coding errors have to be resolved before the programmer can run the code or add new functionality to the program. If the error message does not give the (kind of) information the programmer needs or uses terminology the programmer does not understand, it becomes a lot harder to fix the error it is reporting. This thesis aimed to enhance the understanding about why the error messages of the Python programming language fail to be understood by novice programmers. An experiment was performed where novice programmers constructed how they thought the error messages should look like when encountering errors often made by novice programmers. This thesis also introduces the Error Message Component Framework, a framework which can be used to determine the different components of information an error message contains and the structure of these components. This framework was used to compare the original error message to the custom error messages created by the participants of the experiment, to investigate if the original message matched the novice programmer's information expectations. This thesis concludes that even though some errors are often made by novice programmers, their resulting error messages may not targeted towards this group based on the terms used in these messages, the component of information these messages contains and the lack of precision of the information.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Ir. Felienne Hermans, Faculty LIACS, Leiden University
Committee Member:	Prof. Dr. Marcus Specht & Dr. Mauricio Aniche, Faculty EEMCS, TU Delft

Preface

First of all I want to thank my fiancee, who showed more patience and support than I could have ever expected from her. I hope that I have the rest of my life to repay you for it. I also want to thank my supervisor Felienne Hermans. She was a great inspiration and a great mentor. She knew when to slow me down when I wanted my ambition to run wild, how to focus my efforts into something feasible, and how to motivate me when I had not realised yet how far I've. Though my time as a master student will end, I hope I'm still allowed to join the weekly meetings of the PERL group and maybe collaborate on opportunities to improve Computer Science education.
Lastly I want to thank the group that helped me in creating this thesis: my students, current and past, from Atheneum College Hageveld. Their enthusiasm, mistakes and questions inspired and motivated me to do the research as documented in this thesis. Without them I was just a nerd who was interested in Computer Science, because of them I am a teacher.

Bart Heemskerk
Delft, the Netherlands
October 9, 2020

Contents

List of Figures

Chapter 1

Introduction

Learning to program is not a easy thing to do. Besides learning to read and understand a computer program in a programming language, a novice programmer has to learn how to create code in the same language. Writing code is of itself a daunting task, as small mistakes can result in a program that will not compile or run. If this mistake results in erroneous code or an error during the execution of the program, the compiler, interpreter or runtime environment will create an error message to attempt to communicate what the error is. When the code of a novice programmer contains an error, the error message is their primary source of information for finding the error.

This thesis will investigate if the error messages that novice programmers often encounter when programming in Python do contain the information or type of information the novice programmers expect. This thesis will do so by analysing the error messages which correspond with error often made by novice programmers, and performing an experiment where novice programmers encounter and try to solve certain errors within multiple pieces of code. In the debugging experiment, the novice programmer will give feedback on the presented error message and create an error message of his/her own.

1.1 The Evolution of Programming Languages

Error messages are a form of feedback which programmers often take for granted when coding. However, the first programming languages did not have this form of feedback. In order to understand the current state of programming languages and their error messages, the history of different generations of programming languages are summarized.

Programming is writing lines of code in order to create software or automate a certain process. Lines of code written in a programming language are statements which instruct the computer how to alter their state. Nowadays this code is in the form of text in a stored file and follows the strict specifications of a programming language such as Java, Python or Scala, but this was not always the case.

The Collosus computer is considered one of the first programmable computers[30]. This computer was not programmed using a stored program, but by altering switches and plugs. The first stored programs were written in Machine Languages, which were later dubbed the First-generation programming languages. Programs in machine languages were written in a numeric form, meaning that a (binairy) number represented an action or a memory location. These programs could be executed by the hardware directly and did thus not require any translation effort for the computer. It did require a lot of translating effort for the programmer, who had to know which number represented which action and what the values in certain memory positions represented. This created problems when the code contained an error or did not work as intended. The cause of the error was hard to find because of the translation-barrier.

Second-generation programming languages, more commonly known as Assembly Languages, solved part of problems of the first generation languages by using short instruction (such as ADD for adding two values in standard registers, and INC to increment the value in a register) and creating names for certain registers. These instructions could easily be translated to machine languages. Some of the problems of the First-generation languages persisted in the Second-generation languages. The language was a bit more legible, but if the code contains an error it is still hard to find because of the amount of instructions required to do “easy” calculations.

The Third-generation languages were the first languages that did not resemble the machine language that processors use. Third-generation languages use compilers to translate code, that is more similar to natural language, to instructions which can be used by the processor. It is easier to spot errors in these kind of languages, because the languages are closer to natural language than the previous generations. When a piece of code created using a Third-generation programming language contains an error, it might have one of three consequences: the code will not compile because it violates the rules of the programming language (a compile error), the program will crash while it was running because it want to perform an illegal action such as dividing by zero (a runtime error), or the program will run without problem but will not function as intended (a logical error). When a compile or runtime error occurs, then the programming or runtime environment will give feedback to the programmer in the form of an error message.

1.2 The Known Imperfections of Error Messages

Each new generation programming language seems to aim to be more accessible for programmers. In order to achieve this accessibility, the error messages have to evolve along with them. Ideally the error messages should be even more accessible and understandable than the programming language it is reporting about for novice programmers, in order to form the bridge between the language the new programmer understands and the language they have just learned or have yet to learn. According to research, this is not the case.

Barik et al.[1] discovered that error messages are as hard to read as source code and that novice programmers spend a significant time reading an error message. This is a problem, because if the programmer has not mastered the language then it is likely that have a difficulty understanding the feedback. Prather et al.[27] confirm this suspicion. Two students were so unfamiliar with the programming material that they were unable to fix the error, even when the error message was enhanced. Another problem is that responding to the error message might not fix the cause of the error. Kohn [21] was unable to create a metric for determining if an error was fixed, because the programmers sometimes addressed the error message in another way than fixing the actual error. Some deleted the code that caused the error or rewrote it in a way that they were familiar with, which made the error message disappear but did not fix the initial error.

The effectiveness of error messages have been the center of multiple studies, other than the above studies. For example, Nienaltowski et al.[24] investigated if additional information aided novice programmers in message comprehension or improved the time in which they resolved the message. The additional information failed to do so. Denny et al.[9], Becker[4] and Pettit et al.[26] investigated the effectiveness of enhancing the error message, but got conflicting results. This may indicate a lack of uniformity in their approach or in the way (enhanced) error message are constructed or displayed.

This raises the question: How were those enhanced error messages constructed? What kind of additional information did they provide compared to the original? Why did one enhanced error message improve results [4] while the other did not [9, 26]?

Whether enhancing error messages is helpful for novice programmers or not, the effort itself is an indicator that the original error message does not always help novice programmers to solve the error in their code. It is possible that the original language used in the error message uses to much jargon or unknown terms, which makes it hard to understand the feedback given. This may be especially true if the programmers first language is not English, which may hinder the comprehension of the message because of a language barrier.

Whenever a person tries to learn a new language, a programming language or a spoken language, they will encounter words or terms that are unknown to them. It may be problematic if novice programmers encounter terms they do not understand in error messages, because it may hinder their debugging process. The error message often is their first and only feedback about an error in their code. Novice programmers also lack the information gathering tactics more experienced programmers use, like consulting the API or Stackoverflow. If these unknown terms are prevalent in Python error messages, it may demotivate the novice programmer to try and fix the error themselves since they will not understand the error message anyway. This degrades the programming learning experience into a problem that has to be solved, which is troublesome if the novice programmer is learning Python as part of a school curricula.

1.3 Research Questions

Writing correct code can be a difficult task for novice programmers, as they may not be fully familiar with the language they are writing in and relatively new in translating problems to code. This unfamiliarity with the programming language will result in code error, which will result in error messages from either the compiler/interpreter or the runtime environment. In order to investigate if these error messages contain the information that the novice programmer expects, this research will investigate types of information in Python error messages and the information need of the novice programmers. This research will investigate the error messages of the Python programming language. Python is a popular language for learning programming and has been used for many introductory programming courses[32, 19, 37].

Though Python is a popular programming language for beginning programmers, novice programmers still seem to have a hard time understanding Python error messages. Enhancing the error message has not proven to be an effective enough aid for novice programmers. This thesis will therefor attempt to answer the following question:

> **RQ**: Do Python Error Messages match the expectations of novice users?

This question is divided into these sub-questions:

> **RQ1**: What components do Python error messages consist of?

Before information expectations can be compared to the actual information in error messages, a method is needed to identify components of an error message and what kind of information these components contain. **RQ1** will aid in the search for methods that are currently used for programming error message construction. Are there any existing methods for identifying certain types of information? Answering this question will reveal what kind of information the Python error messages contain, thus creating a baseline to compare expectations against.

> **RQ2**: What information do novice programmers expect when encountering certain errors?

Error messages that do meet the expectations can still hinder the debugging process, because programmers may not have all the information they think they need to resolve the error. This may apply especially true for novice programmers, who do not have much experience evaluating the error message and its usefulness.

Answering **RQ2** requires insight into the expectations of the novice programmer. If this were to be done by asking novice programmers by interviewing them about what components of the error message they did or did not use, it would only test the usefulness of the error message without revealing the expectations of the programmer. By asking the novice programmers to write down their own version of an error message for an certain error, it is possible to determine the type and amount of information they are expecting using the same method used for analysing the original error messages. This may be asking much of the novice programmer, because they may not have any expectations and do not know how to construct their own error message. Supporting questions will be asked to help the novice programmer construct their own message based on what they understood or did not understand about the error message presented to them.

> **RQ3**: Do error messages in Python often encountered by novice programmers contain unknown/unclear terms?

A summary of problematic terms can be created by asking the participants to label all the terms they did not understand. Those terms are then also presented in their native language, in order to investigate if the language barrier is the problem of the phrasing. After revealing the cause of the error the necessity of the term is determined by looking at the error message that the participants created. If the amount of occurrences of the term decreases, then the presence of that term might be unnecessary or problematic.

The contributions of this work are:

- A model to determine and compare information type and quantity (and structure) for error messages
- Suggestions for improvements in Python Error Messages for the usage by novice programmers
- A discussion about if and how error messages should be incorporated into Python curricula

Chapter 2

Related Work

This research aims to investigate if the information expected by novice programmers matches the information provided by Python error messages. An understanding of how error messages are created is needed before the components of the error messages can be determined. If there are any guidelines that state that certain components should be present in an error message or that say anything about how error messages should be structured, then that would greatly benefit this research. The results of this search will be shown in Section 2.1.

In order to research the error messages relevant to novice programmers, the errors have to be made by a novice programmer. Novice programmers will make some type of errors more than other another type of error. Section 2.2 will outline the research on the types of error made by novice programmers.

That novice programmers tend to struggle with error messages has become common knowledge. Several attempts have already been made to improve the error message in a way that benefits the novice programmer. Section 2.3 will show several attempts to improve the debugging effectiveness of the error message. The attempts include showing how other programmers have fixed the error and enhancing the error message for better comprehension.

2.1 Constructing Compiler Error messages

The construction of error messages has been the topic of previous studies in the field of compiler construction. One of the earlier works is of J.J. Horning [18], who wrote how the compiler should give feedback to the user. Horning stated that error messages should be user-directed, source-oriented, specific, complete, readable, restrained and polite, and should localize the problem. These characteristics are inspired by human dialogue. Horning put emphasize on the messages being restrained and polite, because the user must be the master and the compiler the servant.

Shneiderman [34] made five recommendations after conducting several controlled experiments on Cobol compiler syntactic error messages with undergraduate novice users as the demographic. His recommendations were:

- to increase attention to message design,

- to establish quality control,
- to develop guidelines,
- to carry out acceptance tests, and
- to collect user performance data.

He also created a summary of what a compiler message should and should not contain, along with other considerations. Shneiderman states that compiler message should be brief, positive, constructive, specific, comprehensible, and should emphasize user control over the system.

A more recent article on the construction of compiler error messages is written by Traver [36]. Traver recognizes the barrier that poorly designed error messages create for novice programmers, where misunderstanding does not create a productivity or quality problem but a problem with learning and teaching a programming language. Traver proposed eight principals for designing compiler error messages, along with an in-depth explanation of why each principle is important.

Traver also added to the principle of *proper phrasing* that the error message should be nonantrhopomorphic, thus should not contain “I can‘t ...” and other constructions that gives the compiler human characteristics, based on related work. Traver does question if this is the case with compiler message and thinks this topic deserves further research. Traver’s principle of *Extensible help* is the result of making the principles of *brevity* and *constructive guidance* compatible, in other words having short yet explanatory error messages. Traver describes an error message that is divided in levels. The first level should be a short message that should suffice most of the time. If not, a next level is to add some explanation or examples to assist the programmer. Another proposed level is where the compiler suggests a list of potential corrective actions.

Barik synthesizes five error message design guidelines based on his research on how error messages are presented and how developers read and visualize error messages[2]. His guidelines are based on previous research such as the recommendations of Horning [18], as well as his own research and the psychology theories as presented Dean[8]. Barik’s guidelines are:

1. Implement rational reconstruction for humans, not tools.
2. Use code as the medium through which to situate error messages.
3. Present rational reconstructions as coherent narratives of causes to symptoms.
4. Distinguish fixes from explanations.
5. Give developers autonomy over error message presentation.

Barik’s guidelines favor a human-centric approach, where the compiler has to take the needs of the developer into account instead of giving feedback that is correct but does not explain why an error occurs or not with a formulation that the developer understands.

2.2 Errors Made by Novice Programmers

Several publications are dedicated to errors in code created by novice programmers and where they are made most often. Pritchard [28] analysed 640.000 submissions to the introductory Python course website CSCircles[29] and 440.000 compile events collected by the BlueJ Blackbox data collection program[1]. He showed that the most made Python errors are Syntax errors with the error message ‘invalid syntax’, which occurred almost twice as much as the Name error which was the second most made error. The reason for this is that ‘SyntaxError: invalid syntax’ is a generic error message and is shown for a lot of different code defects. The most common Java errors are “can’t find symbol”-errors, with three variations of this kind of error within the five most frequent errors. This type of error is caused by using either a variable, function or class that is not defined and is similar to the Name error in Python.

Kohn[21] did a more in-depth research on what caused the different kind of error messages caused by errors made by novice programmers. He discovered that about 30% of the error messages were caused by minor errors, such as missing a closing bracket, a comma, or a colon, or misspelling the name of a variable. Kohn also noticed that the error message was not always correct, because of where the error was located. As an example he showed a code where the student used the wrong method of writing a floating-point number (using a comma instead of a full stop). This is a syntax error, but the compiler recognise it as a type error because it was used as an argument for a function call and the comma makes it seem like an additional argument is added.

2.3 Improving Error Messages

The usage of compiler and runtime error messages has been studied thoroughly [1, 3] and several papers have developed tools to enhance the error message with natural language explanations [4] or suggestions on how to fix the error [15]. These solutions show a problem about the error messages: they are non-intuitive and need explanation to understand. Reading error messages is as difficult as reading source code and takes a substantial time when debugging [1].

Simply enhancing the error message is not enough, as demonstrated by Denny et al[9]. Becker[4] was able to improve on these results, showing significant improvement in repairing bugs for some types of error messages. He does not go into further detail why enhancement works for some errors and not for others, which could be the result of a lack of understanding about why there were differences in the results. Becker’s results were not reproducible by Pettit et al. [26]. Pettit et al. argues that the difference in results might be because they did not enhance all their compiler error messages and because there are differences in the way Becker and Pettit displayed their enhanced messages.

Danny, Prather and Becker[10] were able to reproduce an improvement in result by using enhanced error messages , using a recent collection of guidelines in programming error message research[5]. This collection of guidelines was the result of the ITiCSE 2019 Pro-

[1] `https://bluej.org/blackbox/`

gramming Error Messages Working group[2] and contain the guidelines and characteristics as described in Section 2.1. Danny, Prather and Becker improved the readability, effectiveness and perceived usefulness of four error messages in the C programming language by using full sentences instead of the original, cryptic error message.

A recent paper on enhancing error messages was published by Kohn and Manaris[22], where they enhanced Python error messages in the TigerJyton editor[3]. Although they show great consideration for the placement of the error message, they do not elaborate on the error message construction. In addition to enhancing the error message Kohn and Manaris added a debugger which uses a data visualizer, but through a survey they discovered that this was feature that novice programmers did not use very often.

Nienaltowski et al.[24] have tried to gain insight into what type of error messages work for novices. They concluded that longer error messages do not mean the novice programmers have a better comprehension of the error when compared to their comprehension of regular compiler error message representation. They also concluded that the placement and structure of the information is more important, which might explain the difference in results by Becker and Pettit et al.

Hartmann et al.[15] used a different approach for aiding novice programmers. They designed *HelpMeOut*, a social recommendation system that aids debugging by suggesting solutions that other programmers have used in the past. They implemented the system for two programming languages and were able to give useful fixes for 47% of the errors after a collective 39 hours of programming by novice programmers. Hartmann et al. do note that may not be suitable for teaching purposes, as the Computer Science teachers who got a demonstration of *HelpMeOut* feared that students who procrastinate may benefit from fixes added by students who start early.

[2] `iticse19-wg10.github.io`

[3] `http://jython.tobiaskohn.ch`

Chapter 3

The Error Message Component Framework

The overall aim of this thesis is to determine if Python error messages match the expectations of novice programmers. Before any matching can be done, a method is needed to compare the error message to the expectations. This thesis requires that method to analyse the error message in a way that specifies what kind of information certain components of the error message has.

Characteristics	**Explanation**
Clarity and brevity	The error message should be clear in their meaning and use the minimal amount of words to do so.
Specificity	The error message should specify what has gone wrong.
Context-insensitivity	The same error should result in the same error message independent of the context.
Locality	The error message should report the error as close to where the actual error occurs.
Proper Phrasing	The error message should have a positive tone (thus avoid words like illegal and invalid), should contain constructive guidance, and should use the language familiar to the programmer.
Consistency	Decisions made concerning the error message should apply to all error messages.
Suitable visual design	The error message should have a physical format and use colors or different fonts.
Extensible help	The error message should be able to layer the amount of information and help it gives to the programmer.

Table 3.1: Desirable characteristics for error messages as proposed by Traver [36]

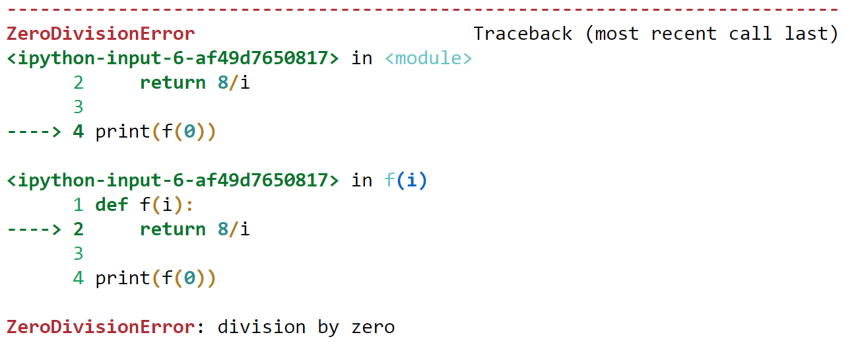
The researches of Horning, Scheiderman and Traver [18, 34, 36] attempt to define what kind of characteristics (compiler) error messages should have. Some of these characteristics are that error messages should be clear but brief, specific, and that the same error should result in the same message in different contexts. The characteristics stated by Traver (shown in Table 3.1) could be used to compare the original error message to the error messages of the particpants, but this would not answer the questions this research intents to answer. The characteristics do not give a clear insight in what components the error message has or what parts of the error message is components is responsible for a specific kind of information. A more in-depth method of analysing the error message is required to compare the error messages.

This chapter presents the Error Message Component Framework (EMCF), a framework to determine the structure of different types of information in an error message. The different kinds of information that the model distinguishes is explained in Section 3.1. Section 3.2 explains how the distinction of information is used to determine the structure of the error message. The uses and limitations of the framework are discussed in Section 3.3 and 3.4 respectively.

3.1 Categories and Elements: The Components of an Error Message

Most information in programming error messages (either compiler or runtime) are based on two main sources: the code in which the error occurred and the actual error. To make a clear separation between these sources, the EMCF distinguishes two categories in error messages: Context and Feedback.

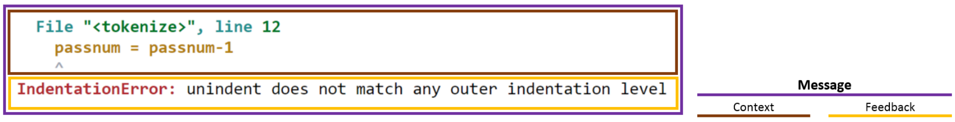
Context is information about where the error is made in the code. This is the information within the code that changes the most when making the same error in a different code. The word context is used over location because of the way runtime errors are often shown. If the runtime error is discovered in a scope other than the main scope, the context will show a trace of all user-defined-function calls made that caused the program to end in that scope. An example for this is seen in Figure 3.1, where calling *f(0)* in the main scope causes an error in the scope of the called function *f*.



```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-af49d7650817> in <module>
      2     return 8/i
      3 
----> 4 print(f(0))

<ipython-input-6-af49d7650817> in f(i)
      1 def f(i):
----> 2     return 8/i
      3 
      4 print(f(0))

ZeroDivisionError: division by zero
```

Figure 3.1: A runtime error concerning multiple scopes

Feedback is the information in the error message about what (type of) error was made. Feedback is one or more lines of text that tries to tell the programmer what is wrong about the code or what went wrong during the execution of the code.



```
  File "<tokenize>", line 12
    passnum = passnum-1
    ^
IndentationError: unindent does not match any outer indentation level
```

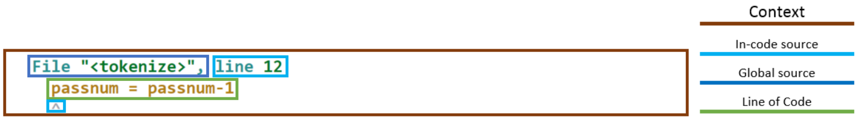
Figure 3.2: The error message divided by category

Figure 3.2 shows how an error message for an incorrect unindent is divided into categories of information. The blocks of information are colored for easy identification.

The two categories give a global idea of how an error message is structured, but are very generic. Without further specification each error message with context above the feedback would seem structurally the same. Each category is therefor further specified into elements.

3.1.1 The Elements of the Context

The context can be divided into three elements: Global source, In-code source, and Line of Code (LoC). Any information about the scope or the file in which the error occurred is part of the Global source. The In-code source information are the text and symbols that indicate where in the global source the error has occurred or was discovered. The LoC are copies of lines from the source code.



```
  File "<tokenize>", line 12
    passnum = passnum-1
    ^
```

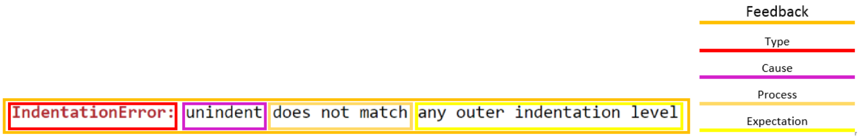
Figure 3.3: The error message context divided into elements

If the elements of the Context are determined for the error message in Figure 3.2, it will result in the division as seen in Figure 3.3. *File "<tokenize>"* gives information on the Global Source of the error, because it refers to a specific file. *Line 12* refers to a source within the code, which makes it part of the *In-code source*. *passnum = passnum - 1* is a copy of the code and therefor a LoC. The symbol under the LoC (^) indicates a location within the Line of Code thus in the source code, which makes it part of the *In-code source*.

3.1.2 The Elements of the Feedback

Although the feedback is often one line of text, the type of information in this one line can vary greatly between different errors. Sometimes the difference is because of the phrasing. For example: "Missing a)" and "Expected a)" contain the same information, but the first line states what caused the error while the later expresses what the compiler or interpreter was expecting during a certain part of the translation.

The EMCF divides the feedback into four elements: Type, Cause, Process and Expectation. The Type is classification of the error. For Python error messages this will be one of the error types as described in Appendix A. Everything describing the source of error is part of the Cause. Process tells the programmer what the compiler/interpreter/runtime environment was doing or trying to do when the error occurred. Expectation is the part of the feedback where the message tells what the compiler/interpreter/runtime environment was expecting when the error occurred.



```
IndentationError: unindent does not match any outer indentation level
```

Figure 3.4: The error message feedback divided into elements

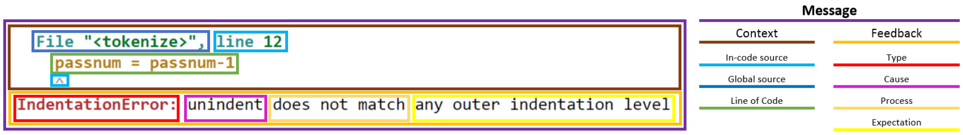
Figure 3.4 shows how these elements of feedback can be found in an error message. *IndentationError* is one of the error types found in appendix A. The error was caused because there was an unindent in the code, thus *unindent* is the Cause. The interpreter was trying to match the indentation of the line to a previously used indentation level, which makes *does not match* part of the Process. Lastly, the Expectation was that the indentation was equal to *any outer indentation level*.

3.2 Using the Components to Determine Structure

By determining which parts of the error message contain a certain kind of information, it is possible to see how information is structured. However, it does not provide a reproducible way to compare the structure. This section describes how the division of information can be used to document and quantify the structure of the error message.

3.2.1 Building a Structure Tree

The identification of information makes it possible to create a tree based on the categories and the elements. The construction of the tree will be shown based on the example of Figure 3.5.



```
  File "<tokenize>", line 12
    passnum = passnum-1
    ^
IndentationError: unindent does not match any outer indentation level
```

Figure 3.5: The error message divided by type of information

Creating the Root

The message as a whole will be represented as the root node of the structure tree. The nodes of the tree are colored with the same colors as used in the identification of the information.

Branches of Categories

The first division in information was based on the categories context and feedback. The categorie of the first line is determined and a node of that categorie is added to the tree. This node is placed right of the root with a connection to the root. If the next line in the error message is of the same categorie as the current, then no new node will be created and the next line becomes the new current line. If the next line has a different categorie as the current line, then a new node of the other categorie will be created under the last created node. This node is connected to the root node. The next line will once again become the new current line. This process is repeated untill the last line of the error message is reached.

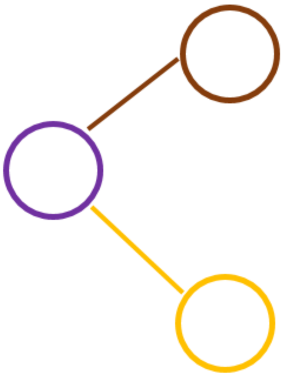


Figure 3.6: The structure tree representation of the error message and categories

Figure 3.6 shows the application of this process on the example. Because the error message only changes once of category when examining the message line for line, between line 3 and 4, it results in two added nodes.

Branches and Leaves of Elements

Adding the elements to the tree also starts at the first line. The lines are read left to right, corresponding to how the English language is written/read. The first element of the line is added to the tree as an element node right of the category node corresponding to that line. A connection is made between the element node and the category node. For each following element in the line a new node is created right of the last created element node (with a connection between them) until the end of the line is reached. This process is repeated for each line, where each new branch from the categories is placed under the last branch.

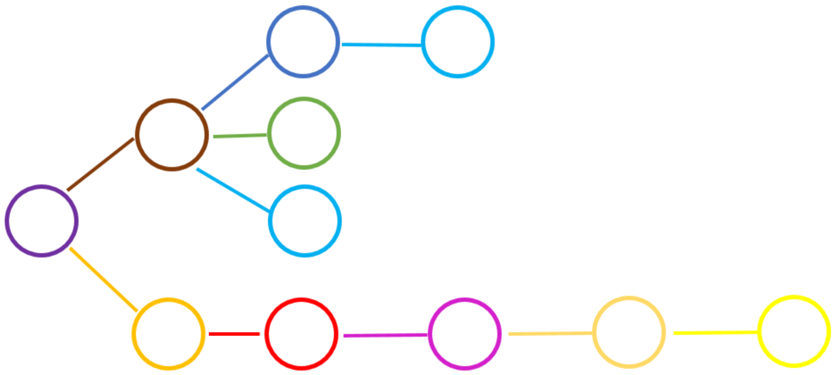


Figure 3.7: The structure tree representation of the full error message

The full tree for the unindent error is shown in Figure 3.7. The three branches connected to the context node represents the three lines of information in the error message that are part of the context.

3.2.2 Adding Weight to the Structure Tree

The tree created in Figure 3.7 shows how the error message is structured. What it currently lacks is a form of quantification. A line of feedback containing 10 words can not be distinguished from a line of feedback containing 40 words if they have the same order and amount of elements, even though they clearly hold two different amount of information. A value has to be added to each node, where the value represents how much information each element block contains.

The Amount of Information in the Elements

Ideally the quantification represents how much information the reader receives from the message, because how much information one gets from a text is reader-dependant. However, this kind of quantification limits how objectively trees of different error messages can be compared. If a reader understands a certain kind of error A but has no experience with error B, then the quantification would probably vary wildly even thought error A and B may be structurally similair.

The measure used for the model is the amount of words that are part of the element. This method guarantees that two different persons come to the same number when given the same division of elements when looking at the text in the error messages.

Most error messages also use symbols to provide information. The compiler error messages have an arrow under the line of code in the error message, indicating where the error was discovered by the compiler/interpreter. Runtime error messages have an arrow left of the LoC that caused the error. These symbols will be counted as 1 word, because the information of the symbol is equivalent to “here”.

Lastly the value of the LoC’s has to be set. The content of the LoC varies depending on the code in which the error occures. If the quantification of the LoC would be dependant on the code, then a difference in overall quantity of information in an error message could be attributed to working with different code. This is not desireable because it could cause error messages with the same error cause to not be similair to themselves. A fixed amount is therefor used for LoC elements.

Table 3.2 shows a possible verbatim of the lines of code when written down in natural language. Although the verbatims will become longer depending on the variants, it appears that often a minimum of four words in natural language is needed to express what is happening in a line of code. The “amount of words” for LoC’s will therefor be set to four.

Line of Code	**Line of Code as a Sentence**
print(<*parameters*>)	print the value(s) <*parameters*>
<*variable*>= <*value*>	set <*variable*>to <*value*>
if <*condition*>:	if <*condition*>is true
else :	if all conditions fail
while <*condition*>:	while <*condition*>is true
for ... in <*list/range*>:	for each value in <*list/range*>
<*list*>.append(<*value*>)	append <*value*>to <*list*>

Table 3.2: Possible verbatim for lines of code

The quantification of the whole error message (The root of the structure tree) starts with the elements of the message. The value of each non-LoC element node in the structure tree becomes the number of words in the corresponding element of the error message. For example: if a in-code source node represents the in-code source element *line 14*, then the value of the node becomes 2. Each LoC node gets the value 4 for reasons stated before. Applying this process to the tree in Figure 3.7 results in the values as shown in Figure 3.8.

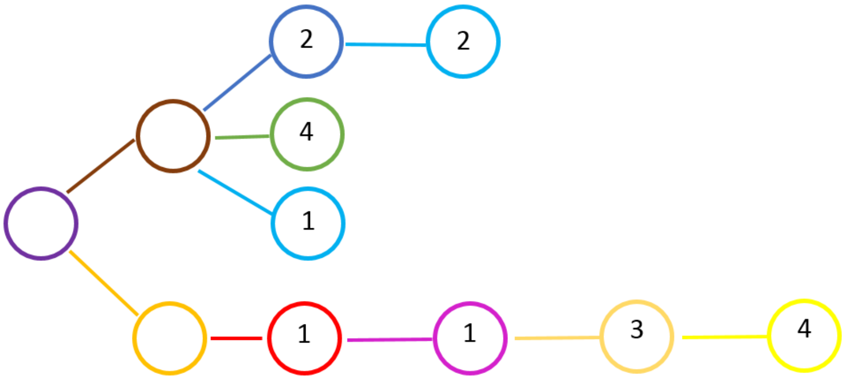


Figure 3.8: The structure tree with the amount of words in each element node

Propagating to the Categories and the Message

After quantifying the elements, it is possible to propagate the values to the higher levels of abstraction: the categories and the message. The value of each category node becomes the sum of values of all element nodes that have that category node as its root. The value of the message is determined in the same manner, but summing up the values of the category nodes instead. This will result in a structure tree where each node is assigned a value representing the amount of "words". The final result of applying the framework on the unindent error message can be seen in Figure 3.9.

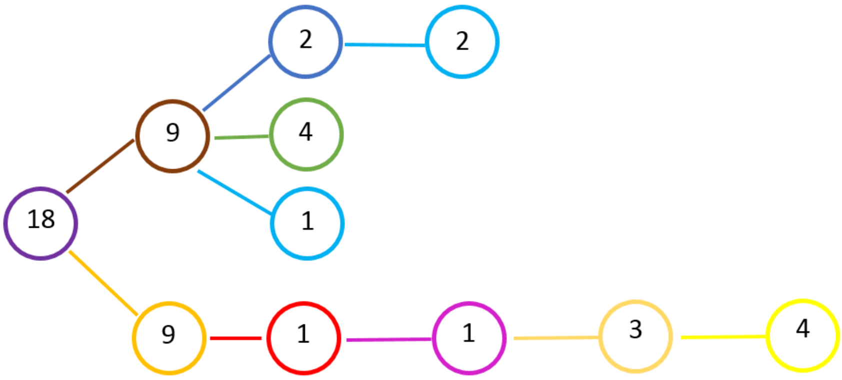


Figure 3.9: The structure tree representation with information quantification

3.3 Benefits of the Framework

3.3.1 Eye Tracking Data

Barik et al.[1] used eye tracking to determine how much time novice programmers spend reading the error message. The error message is discussed as a whole in their research. In order to gain a deeper understanding about how novice programmers use error messages, a more precise method of identifying where the programmer is looking is necessary. If the EMCF is applied to an error message, it creates the possibility to talk about specific parts of that error message based on the element they are labeled as. Developers who read error messages do not read error messages as a whole but read individual words, symbols or lines of code. Using the components of the EMCF, it is possible to say "The developer reads the Type of the error message" instead of "The developer read the error message".

The EMCF also creates an opportunity to be more exact when evaluating eye tracking data. Barik et al.[1] evaluated how long developers were looking at the error message as a whole. Using the identification of the EMCF, it is possible to divide the evaluated error messages into categories and elements. The amount of time a participant looks at a category or an element can tell how the participant is dividing its attention between the different kinds of information provided by the error message.

3.3.2 Proportions of the Elements

The EMCF results in an amounts of information per category and elements. These amounts can be used to compare how much information is dedicated to a certain element or category. The proportions indicate the distribution of different types of information. The proportions can be determined in three ways: Elements-to-Category (E:C), Categories-to-Message (C:M), and Elements-to-Message (E:M). Each proportion shows the focus of the error message on a different level. The E:C-proportion shows what element each category focuses the most on. The C:M-proportion shows if the message focuses on where the error happened or what the error is. The E:C-proportion and C:M-proportion can be be used to determine the E:M-proportion, which indicates the element of information the overall message focuses on.

Type of Component	**Component**	**# of Words**	**E:C-Proportions**	**C:M-Proportions/ E:M-Proportions**
Message	*Message*	18	X	X
Categories	*Context*	9	X	0.5
	Feedback	9	X	0.5
Elements	*Global Source*	2	0.222	0.111
	In-code Source	3	0.333	0.167
	Line of Code	4	0.444	0.222
Elements	*Type*	1	0.111	0.055
	Cause	1	0.111	0.055
	Process	3	0.333	0.167
	Expectation	4	0.445	0.222

E:C-Proportion: The proportion of a Category dedicated to this Element of information
C:M-Proportion: The proportion of a Message dedicated to this Category of information
E:M-Proportion: The proportion of a Message dedicated to this Element of information

Table 3.3: Proportions of component types present in the error message in Figure 3.2

3.3.3 Determining Differences in Structure

Nienaltowski et al. concluded that the structure of the error message is an important part of how novice programmers understand the information [24]. The structure tree of the EMCF models information in an error message in a way that makes it easy to interpret how this information is structured. Using the tree as shown in Figure 3.9 as an example, it easy to see that the error message first focuses on where the error is before it talks about what the error is. It is also easy to read that the feedback starts with the Type of error and consist of one line only.

The structure tree also enables structure comparison between two different error messages. The similarity of two structure trees can be compared based on the criteria that researchers may want to evaluate. Some of these criteria can be:

- **Do the error messages follow the same categorical structure?** This can be evaluated by examining the category nodes and the order in which they are connected to the root.

- **Does the context always follow the same structure?** This can be determined by looking at the context node(s) and determining if there is a one-to-one match of each branch.

3.4 Limitations

3.4.1 Non-text Methods of Communicating Errors

The Error Message Component Framework currently models one error message for one error, assuming this error message is fully in text. More advanced programming editors and IDE's have multiple ways of communicating information. The context of the error message may be underlined or highlighted in the source code.

This is problematic because the context of the error message is displayed in a different location in the editor than the feedback. The EMCF expects an error message to be lines of information one after another, not expecting any information communicated outside of these lines. The EMCF builds a structure tree with a root on the left and categories and elements on the right to represent the reading order for the English language. If some part of the information is outside these lines, where should they be anchored? Is it part of the context of the message and therefor part of its context? Is it related to some part of the error message and should therefor be a branch of an element node? How should this be taken into account when comparing structure trees?

Another problem is how this information should be quantified. If a part of a line of source code is highlighted as part of an error message, is the line of code part of the error or just the highlighted section? Should there be a quantifiable difference between one or more highlighted words?

Because these questions are hard to answer, the EMCF is not fit to include non-text methods of communicating errors into modeling the error message. Highlighting erroneous code is a feature not often found in editors designed for or used by novice programmers and will therefor not limit this research.

3.4.2 Multiple Error Messages from Different Errors

When multiple errors are written in a programming language that will be compiled like Java and C#, then the programming environment will show the error message for each error at the same time. The EMCF could be applied to each individual error, but can't model how each message is placed in relation to each other.

If the roots of the structure trees would be connected to create a new root, then this would create the illusion that the information in the messages is related by more than code. Even the relation through code might not hold true if the errors are made in separate files. Forcing the EMCF in its current state to work on multiple error messages from different errors is therefor not possible. This will not limit this research, because Python programming environments often only shown one error message at the time.

Chapter 4

Experimental Setup

This research aims to gain insight in the information expectations of novice Python programmers when confronted with an error in their code. Using the Error Message Component Framework as described in Chapter 3, it is possible to analyse Python error messages based on the type of information they contain and their structure. The next step is to gain insight into the expectations of the novice programmers.

This chapter describes the setup of an experiment where the participant, a novice programmer, is presented pieces of erroneous Python code with an error message to let them experience the error messages currently presented by a Python editor for common novice errors. This experience will help the novice programmer in answering interview questions about the usability and effectiveness of the error message and guide them in creating a custom error message that represents how they want the error message to look like. All decisions made during the construction of the experiment and motivations behind those decisions are described in Section 4.1.

This research will be the first to use this methodology, which means that the time it will take to execute the experiment and the quality of the interview questions is unknown. A small pilot was run to verify both these parts of the experiment. The setup, results and implications of the pilot is presented in Section 4.2

4.1 Methodology

The main goal of this thesis is to investigate if Python error messages are what novice programmers expect from them. Using the EMCF it is possible to understand the components and structure of the Python error messages in their current state. To investigate if these messages meet the expectations of novice programmers, this thesis will let novice programmers create their own custom error messages. These custom error messages represent how they thinks the error message should look like and what information it should contain. Comparing the custom error messages to the original Python error message reveals where the original message meets the expectations and where they do not. Letting a novice programmer create their own error message is a method not previously used in any research as a means to validate error messages. This section will describe the details of the experiment.

This concerns the decision to investigate the error messages of the Python programming language (Section 4.1.1) and general details about the participants (Section 4.1.2).

A novice programmer, when asked about a certain error, will not be able to give information about if an error message was clear or not from memory. It is also highly unlikely that a novice programmer is able to create an error message themselves without context, as they have limited programming experience aid them during the construction of their message. Section 4.1.3 describes the blueprint of an experiment where the participant performs several tasks. Each tasks concerns a piece of code with one error and the corresponding error message. Executing these tasks as well as answering several interview questions, which are described in Section 4.1.4, will aid the participant in giving feedback about the existing error message and creating an error message of their own.

Novice programmers will not create errors about advanced programming concepts, as they recently learned to program. A selection of code errors, which novice programmers are likely to make, has to be created for the experiment. As the editor influences what is deemed an error, the editor will be decided in Section 4.1.6 before finalizing which errors will be used in the experiment in Section 4.1.5

4.1.1 Programming language

In recent years, the popularity of the Python programming language has been growing. The TIOBE[1] Index shows that the amount of sites about the Python programming languages is greater than ever before[35], ranking Python as third. The PYPL Popularity of Programming Language index has Python as the language that has the most tutorial searches on Google[6], surpassing Java and Javascript.

The error messages of Python will therefor be researched, because of the rise in popularity among people who want to learn to program.

4.1.2 Participants

People are learning to program at a much earlier age than they used to. In recent years, lots of research has been done on teaching coding to children using Scratch[2][13, 17, 23]. Scratch is a block based language, where the programmer has to drag and drop new instructions into a coding environment in the form of blocks. A program is created by sequencing blocks to move sprites, change backgrounds, create sounds and more[31].

Computer science education (programs) using Scratch and other block-based languages are usually focused on primary school and middle school[13, 23, 25, 33]. The students in this range are of no interest for this research, because these languages protect the programmer from error messages. If these languages produce error messages, it is most likely because of altered settings meant for more experienced programmers and more advanced projects.

The participants for this experiment will be students of the Atheneum College Hageveld, located in Heemstede, the Netherlands. Hageveld is a VWO-only secondary education

[1] https://www.tiobe.com

[2] https://scratch.mit.edu/

school and teaches Computer Science as one of its optional subjects in their fifth and sixth year. Students following Computer Science are therefor mostly between the ages of 16 and 18. This age-range corresponds with K-12 high school students. The students have followed an introductory course in Python using CSCircles[29], a in-browser Python course for beginning programmers, where they have learned about variables, comments, data-types, using functions, control flow, loops and creating functions.

4.1.3 Task Design

To get data on how novice programmers read and understand error messages, they will have to perform several tasks with the same structure. The task structure will be derived from the setup of Barik et al[1]. Participants will be handed instructions on the process of the research as well as a consent form.

- **Introducing:** The participant is presented with a piece of code with no error, to get familiar with the environment and the method of running the code. A simple task will be given to the participant, to get familiar with the editing en running the code in a notebook.

- **Performing the tasks:** The participant will perform the same sequence of actions, each time with different source code. The time limit for each task is 2 minutes[3]. The tasks are presented in a random order to control for learning effects.

  - **Showing:** The code and corresponding error message is shown to the participant.
  - **Solving:** The participant tries to solve the error (based on the error message). To prevent trial and error, the participant can try their solution only 2 times or until they receive an error of a different nature. An error is successfully resolved when no more error message presents itself.
  - **Evaluating:** The participant will be interviewed about their understanding of the error message and their debugging process. During this interview, the source of the error will be revealed (if not yet found).

- **Finalizing:** The participants will fill in a final questionnaire about basic demographic information and experience.

The school uses class hours with a length of 40 minutes to divide the day, with a break of 20 minutes after the first three class hours and break of 30 minutes after the fifth class hour. 40 minutes will be too short for the experiment to be performed (considering 5 minutes for the task and 5 minutes for the interview). The experiment will therefor take an hour. The number of tasks will be adapted to this time-frame.

[3] This was initially 5 minutes, see section 4.2.2

4.1.4 Interview questions

The participant will be interviewed about their understanding of the error message. Using a copy of the editor and error message, the participant will be asked the following about his/her interaction with the code and the error message:

- Where do you think the error is in the code? (indicate by drawing the circle around the location)
- How did you get to this conclusion?/Which information did you use to get to the pinpointed area?
- Which words/parts of the error message did help in this process? (underline these words in blue)
- Which words/parts of the error message did you not understand? (underline these words in red)

After this point a translation of the error message will be presented to the participant if there is any part of the error message that the participant did not understand. All words that are not file-specific or related to the type of error (see Appendix C) are translated into Dutch. The participant is then asked the following question:

- Is the error message more clear in Dutch? (do you understand the words that you previously did not understand?)

The cause of the error will be unveiled after these questions are asked, if the participant has not figured out the error at this point. The participant will then be asked to think about what information the error message should contain to locate, understand and correct the error. The interviewer might use the information that the participant has already given to stimulate the participant, for example by repeating what the participant thought was useful about the error message and if they would use reuse the phrasing or the information that they got from it.

- For this error *(repeat error)*, what information do you want to have to know what the error is and how to fix it?
- How would you have preferred the error message to look like?

At the end of the experiment, the following information will be asked to get some insight into the basic demographics of the participant:

- Gender
- Age
- Specialization
- Years of programming experience (if any)

- Grade for their Python test
- Average grade this year for the English subject

The Influence of the Experiment on the Interview

The participants are asked to create their own error message for a certain error after they have seen the current error message for that error. Seeing this error message may influence how they construct their own message. Instead of an error message that is created from scratch, the participant might reproduce the original error message with a few adjustments.

This is a risk that has to be taken, because the participants are novice programmers. Experienced programmers have encountered multiple error messages in multiple languages and probably know what information they need. They are therefor more likely able to construct a preferred error message without any other context other than the cause of the error. Novice programmers do not have access to this kind of experience. Their lack of experience and programming knowledge may hinder their ability to create their own error message. Although showing the original error message before the novice programmer can construct their own error message can be considered a form of suggestive questioning, the lack of answer is considered more harmful to the experiment than an personal error message influenced by the original.

4.1.5 Errors

Not all possible types of errors of Python (as listed in Appendix A) are tested in this experiment. Novice programmers are not using the full capabilities of Python programming language. Novice programmers will not use more advanced principles like Object Oriented programming, iterators, or unicode when they recently learned the basics.

The type of errors that remain after excluding the type of errors related to advanced principles are shown in Table 4.1.

Several of these error types are present in the findings of Pritchard[28], who analysed the error distribution from a dataset obtained from CS Circles. After generalizing the 309.710 syntax errors and 333.538 compile-time errors, he determined the error messages as shown in Table 4.2 to be the most common Python error messages.

Error Type	**Cause**
RuntimeError	an error is found during execution of the code that is not part of a subclass of the RuntimeError.
SyntaxError	a line of code code does not comply to the syntax.
IndexError	the code tries to read an index outside of the range of an array.
NameError	a variable/function is used that is not defined
IndentationError	a line of code is (un)indented without cause, or when a line of code with indentation is expected but not given.
TabError	both tabs and spaces are used for indentation in a file.
TypeError	input of the wrong type is given for an operation, a function call or array/dictionary value retrieval.
ValueError	the type of input is right, but the content does not meet the expectations.
ZeroDivisionError	the second argument of a division or modulo operation is zero.

Table 4.1: All types of errors that novice programmers might encounter

Error message	**Amount of occurrences**
SyntaxError: invalid syntax	179624
NameError: name ‘NAME’ is not defined	97186
EOFError: EOF when reading line	76026
SyntaxError: unexpected EOF while parsing	26097
IndentationError: unindent does not match any outer indentation level	20758

Table 4.2: The most common Python error messages

4.1.6 Editor

In order for the participants to edit and run the code to verify if they fixed the error, they need to work in a Python editor. The main requirements for the editor are:

- the editor is similar in use, error message and error message placement as CSCircles[4], the programming environment which the participants used while learning Python.
- the editor can be opened using the recording software of the eye tracker[5].

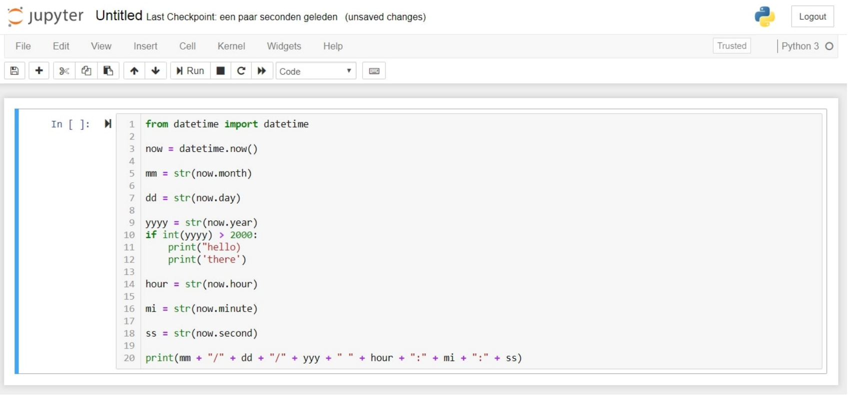
Based on these two requirements, the Python editor used for this experiment will be Jupyter Notebook[6] version 5.7.4. Jupyter Notebook is a web-based platform where Python code can be edited and executed. The advantage of using Jupiter Notebooks is that every task can be a separate Notebook web-page. The recording software of the eye tracker[5] is able

[4] `https://cscircles.cemc.uwaterloo.ca/console/`

[5] The eye tracking data was not used to answer any research questions, see Section 6.3.1

[6] `https://jupyter.org`

to take web-pages as observable media and is therefor compatible with Jupyter Notebooks. The Jupyter Notebooks will be hosted using Anaconda Navigator[7], a localhost platform for performing data science.



```python
from datetime import datetime

now = datetime.now()

mm = str(now.month)

dd = str(now.day)

yyyy = str(now.year)
if int(yyyy) > 2000:
    print("hello)
    print('there')

hour = str(now.hour)

mi = str(now.minute)

ss = str(now.second)

print(mm + "/" + dd + "/" + yyy + " " + hour + ":" + mi + ":" + ss)
```

Figure 4.1: The Jupyter Notebook editor

Jupyter Notebook satisfies both requirements, as it is similar to the CSCircles editor and can be opened using the recording software of the eye tracker[5]. Another reason for using the Notebook editor is that data about the code and the errors made is stored locally instead of on the internet, which protects the information of the participant.

Jupyter Notebook has the additional advantage that when it is saved, the whole state including code, outputs and error messages is saved. This means that when a piece of code with an error is run and saved, it will show the error message when the notebook is loaded again. This prevents having to ask the participant to run the code to show the error message without reading the code first, because the error message is the main subject of this research. If the participants get time to examine the code, they might find the error without looking at the error message.

Error cause	**Amount of occurrences**
Name Error	570
Missing ')'	144
Missing Comma	109
Missing ':'	94
Inconsistent Indentation	64
Unterminated String	50

Table 4.3: The distribution of minor mistakes observed by Kohn

[7] https://docs.anaconda.com/anaconda/navigator/

Implications of the Editor on the Error Messages

With an editor chosen, attempts were made to reproduce error message of the types present in Table 4.1 with explicit attempts to reproduce the messages present in Table 4.2. The error messages in Table 4.2 are caused by novice programmers, but do not point at what error the programmer made. Kohn[21] found that among the 4092 errors his researched data that about 30% were minor, superficial mistakes or misspellings, as shown in Table 4.3. Table 4.3 is used as a basis for designing the tasks for the experiment. In order to get a final list of possible candidate errors for the tasks, all considered errors were tested in the Jupyter programming environment.

The RuntimeError and TabError were excluded after attempting to recreate these errors in the chosen editor. Other types of errors could give different error messages based on the error cause. Table 4.4 contains a description of all errors designed for this research. The code corresponding to each error cause and its error messages are shown in Appendix B.

Task number	Type	Other feedback	Actual cause
1	SyntaxError	invalid syntax	a missing closing bracket
2	SyntaxError	invalid syntax	a missing comma
3	SyntaxError	invalid syntax	a missing colon
4	SyntaxError	EOL while scanning string literal	an non-terminated string
5	IndexError	list index out of range	requesting an array position that's out of range
6	NameError	name 'NAME' is not defined	requesting data from a non-existing variable
7	IndentationError	unindent does not match any outer indentation level	not a correct amount of indentation
8	IndentationError	expected an indented block	no indentation after an if-statement
9	TypeError	'>' not supported between instances of 'float' and 'str'	a parameter of the wrong type in a function call for function *max*
10	TypeError	can only concatenate str (not "float") to str	an operand of the wrong type in an operation (string concatenation)
11	ValueError	could not convert string to float: STRINGVALUE	a parameter of the right type but with an unexpected value in a function call
12	ZeroDivisionError	division by zero	a division by zero

Table 4.4: Summary of all errors considered for the experiment

4.2 Pilot

The setup of the experiment has to be tested to validate its usability and to check if the full experiment can be performed in 60 minutes. The usage of Jupyter notebook in this context has never been done before, thus there is no reference of how much time it takes for the participant to get used to the programming environment.

Although the setup of this research is greatly inspired by Barik et al.[1], the possible programming concepts differ greatly due to difference in programming language and the age of the participants. This means that the time used by Barik et al. for the tasks may be too long, because their tasks uses an OOP-based programming language where the error is a result of inheritance and the code for this research does not use code outside the code on display. The exception in this experiment of task 4, which uses the random library. However the error and resulting error message are not a result any functionality of the random library.

This section will summarize the results of a pilot of the experiment in Section 4.2.1. The implications of the pilot will be discussed in Section 4.2.2.

4.2.1 Results of the Pilot

Three participants of the target demographic participated in the pilot. The participants did not perform all tasks, because it is preferable that students that have the time to perform the full experiment participate in the experiment. Two participants performed task 1, the other performed task 4. None of the participants created an own error message due to time constraints.

All participants performed the introduction. Doing the task as asked in the introduction took 3 minutes at most. When performing a task (showing and solving) the participants took at most 2 minutes to find the error or to indicate they were unable to find the error. One participant requested the task to be ended after indicating that he did not know what the error was or how to solve it. The task was therefor terminated after 90 seconds.

Interviewing the pilot participants gained the insights intended. However, one of the participants had a hard time thinking about what he wants to know to find and correct the error. After further questioning it turns out that he was not confident in his programming skills to determine what is and what is not a good answer, even though there is no wrong answer.

4.2.2 Implications

Based on the results of the pilot, the following changes were made to the experimental setup:

- **Duration of tasks:** Each participant had 5 minutes to find the error in the code. However all participants either found and corrected the error or indicated that they could not find the error within 2 minutes, indicating that 5 minutes is too much time. The participants will therefor be given 2 minutes to execute the task.

- **The amount of tasks:** 12 tasks are too many tasks for the experiment. Each task takes 7 minutes (2 minutes for debugging, 5 minutes for the interview) at most. This would give 84 minutes for performing the tasks, which is more than the ceiling of 60 minutes for performing the whole experiment.

For this reason only non-runtime errors will be included in the experiment (tasks 1, 2, 3, 4, 7, and 8). This would leave out the NameError, task 6, which is one of the most frequent made mistake by novice programmers[21, 28]. In other languages like Java and C# using an unknown variable would be a compiler error. Tasks 6 will therefor also be included, resulting in 7 tasks taking an estimated 49 minutes to perform.

- **Add question about confidence:** One of the participants expressed that he had very little confidence in his programming skills as one of the reasons why he could not find the error in the code. For this reason a question will be added to the basic demographic questions regarding if the participant has confidence that their code will run on the first try using a 5-point Likert scale. Additional explanation can be asked if the result of this question contradicts the data, like a low confidence after finding and correcting every error or a high confidence after having a hard time during the experiment.

Chapter 5

Results

A total of 11 students participated in the experiment, of which 7 were male and 4 were female. 7 participants had chosen a STEM-oriented study program (*Natuur en Techniek* and/or *Natuur en Gezondheid*); 4 participants had chosen a liberal-arts-oriented study program (*Economie en Maatschappij* or *Cultuur en Maatschappij*). The average age of the participants is 17. Three of the eleven participants had programming experience prior to the Python course.

The question central to this thesis is: **RQ** *do Python error messages match the expectations of novice user?*. This question was the root of several sub-questions, which will be answered in this chapter.

The first sub-question is **RQ1**: *What components do Python error messages consist of?* No participant data will be used to answer this question as it only concerns the original Python error messages. The type of information, the structure of the Python error messages, and the proportions of the components will be determined using the Error Message Component Framework (EMCF) as described in Chapter 3. Section 5.1 will describe the full analyses.

The custom error message, made by the participants of the experiment, will be used to answer the second sub-question **RQ2**: *What information do novice programmers expect when encountering certain errors?* Once again the EMCF is used to analyse the error messages. The custom error message are compared to the original Python error messages in Section 5.2, based on their content and their structure.

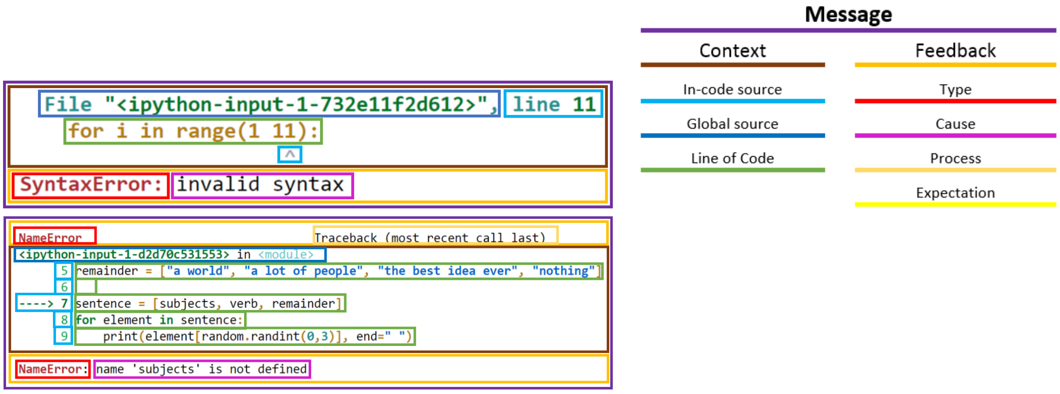
Each participant was asked to label the words in the original error message that they did not understand, which will be used to answer the last sub-question **RQ3**: *Do error messages in Python often encountered by novice programmers contain unknown/unclear terms?* The label terms are then analysed based on which of the original error messages use them and how these terms are used by the participant who labeled them, the results of which are presented in Section 5.3.

The EMCF has been designed with the original Python error messages in mind. The customerror messages, made by the participants, were not created with the EMCF in mind. This resulted in custom error messages that could not be analysed using the original description of the EMCF. Section 5.4 pinpoints where the framework lacked when applied to the custom error messages and how these problems were resolved.

5.1 RQ1: What Components do Python Error Messages consist of?

The Error Message Component Framework divides information into two categories: Context and Feedback. The different elements of the Context are the Global Source, the In-code Source and the Line(s) of Code. The elements of Feedback are the Type, Cause, Process and Expectations. These categories and elements describe what kind of information the error message contains.

Each task of the experiment had a different error with the corresponding error message. Each of these error messages were analysed using the EMCF to determine what element of information each word represented. Two examples of this analysis are shown in Figure 5.1. This analysis is then used to create a structure tree, representing what elements of information are contained in a sentence and in which order, and what category of information each sentence covers. Each node in the tree represents a cluster of words with the same type of information and has a number representing the amount of words used in that cluster. Figure 5.2 shows the structure tree of the two previously used examples.



```
File "<ipython-input-1-732e11f2d612>", line 11
    for i in range(1 11):
                      ^
SyntaxError: invalid syntax
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-1-d2d70c531553> in <module>
      5 remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]
      6
----> 7 sentence = [subjects, verb, remainder]
      8 for element in sentence:
      9     print(element[random.randint(0,3)], end=" ")

NameError: name 'subjects' is not defined
```

Figure 5.1: EMCF applied to the error message of Task 2 and Task 6

The result of the EMCF analysis on all error messages of the tasks and their corresponding structure trees are presented in Appendix D. The structure tree of each error message will be used to inspect if there are any structural similarities between the error messages of the tasks. The results of this analysis is presented in Section 5.1.1. The analysis also show how many words are used to communicate certain elements of information. These numbers can be used to determine how many words proportionally have been used to communicate the elements of each category of information. The proportions and their implications are analysed in Section 5.1.2.

5.1.1 Structural Similarity between Python Error Messages

Applying the EMCF to the error messages of the experiment tasks resulted in twelve structure trees, describing the structure of the error message based on type of information it contains. Figure 5.2 shows two of these trees. The trees consist of a message-node (the

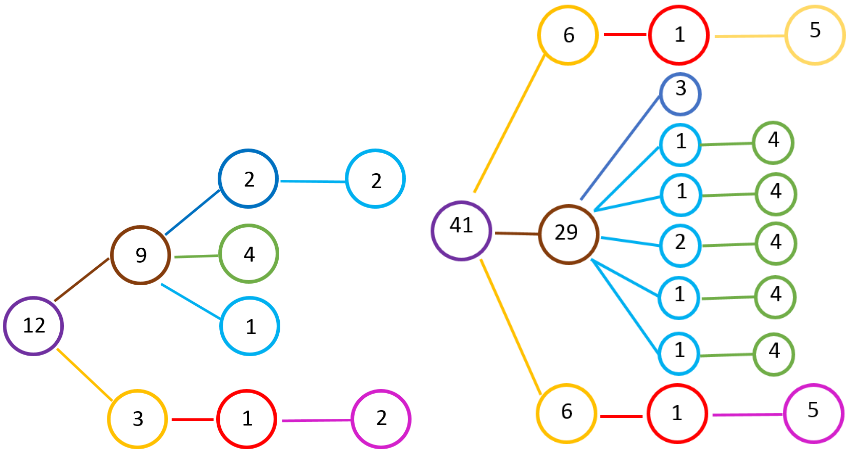


Figure 5.2: The structure tree for the error message of Task 2 and Task 6

purple root), category nodes (the orange and brown nodes directly linked to the message node), and element nodes. Each branch of element nodes, originating from a category node, represents one line of error message.

When looking for structural similarity, the following definitions are used:

- **Same**: Given two nodes from two different structure trees, they are structurally the same if they represent the same type of information, they have the same amount of branches and each branch from those nodes are structurally the same.
- **Similar**: Given two nodes from two different structure tree, they are structurally similar if they represent the same type of information and one node has no branches, or if both nodes have one branch their branches are not structurally the same.

  If one or both node have multiple branches and the node with more branches has no more than twice as many branches as the other node, then there is a subsection of branches from the node with most branches where each branch is structurally similar or the same as the branch with the same position from the other node. These branches are compared top to bottom.
- **Not Similar**:Given two nodes from two different structure tree, they are not structurally similar if they are neither the structurally the same nor similar.

The following questions are answered using the structure trees as shown in Appendix D:

Can error messages be divided into groups based on the structure of the category nodes?

When investigating similarity based on the category nodes, a clear division can be spotted. The error messages in task 1, 2, 3, 4, 7 and 8 all have the context above the feedback (as seen on the left of Figure 5.2). The error messages in task 5, 6, 9, 10, 11 and 12 start with feedback, followed by context and end with feedback (as seen on the right of Figure 5.2).

This difference in categorical structure is caused by the type of error. The first group of error messages (context followed by feedback) are all compiler error messages, whilst the second group (feedback, then context, then feedback) are all runtime errors.

Are there similarities between corresponding category nodes within these groups?

The error messages are separated in two groups based on the shared structural similarity found in the previous question: compiler error messages and runtime error messages.

The compiler error messages all begin with context and end with feedback. The context between all the compiler error messages are structurally the same. They all consist of three lines, where the first line begins with two words of global source followed by two words of in-code source, the second line is LoC and the last line is one “word” of in-code source. The feedback of the compiler error messages have the similarity that they all consist of one line and start with a type. After the type node there does not seem to be any similarity between the feedback of the compiler error messages. The error message of task 1, 2 and 3 are structurally the same, because these tasks have the same error messages.

The runtime error messages start with feedback, followed by feedback and end with feedback. The first feedback node of the runtime error messages are structurally the same, because they all start with the type of error followed by six words of process.

The context of these message have a lot of similarity. They start with one line of global source, followed by four to five in-code sources (the line-numbers) each followed by a LoC. The third LoC of each context start with two “words” of in-code source, one indicator and a line-number. The differences in amount of LoC’s is because of the source code where the error is. The errors in task 5 and 12 are located in the second to last line of code. The line of code where the error occurs is the third LoC in the runtime error message, making it impossible to display the second line of code after the error location.

The later feedback node have the same structural similarity to themselves, but also to the feedback of the compiler error messages. They all begin with a type, but fail to have any clear structural similarity after that. The later feedback of the error messages of task 5, 6 and 9 are most similar, as they all begin with a type only to be followed by a cause. The difference between the error messages is the amount of words used, with task 5 and 6 using 5 words for the cause whilst task 9 is using 9.

5.1.2 Component Proportions of Python Error messages

The structural similarity revealed that Python error messages can be grouped into compiler error messages and runtime error messages. These groups will be used when comparing the proportions of the information.

The proportions of the compiler error messages show a lot of similarity as seen in Table 5.1. All compiler error messages have the same E:C-proportion for the context, which complies with the structural similarities of their context. The messages do have different E:C-proportions for the feedback. The E:C-proportion for Feedback often focus on a single element of information. Task 1, 2, and 3 have the same feedback and they all focus on the cause, task 4 focuses on the process, and task 8 focuses on expectation. Task 7 is the only task that touches all elements of feedback, although not equally.

Proportions	**Task:**	**1**	**2**	**3**	**4**	**7**	**8**
C:M	**Context**	0.75	0.75	0.75	0.6	0.5	0.64
	Feedback	0.25	0.25	0.25	0.4	0.5	0.36
E:C	**Global source**	0.22	0.22	0.22	0.22	0.22	0.22
	In-code source	0.33	0.33	0.33	0.33	0.33	0.33
	Line of Code	0.44	0.44	0.44	0.44	0.44	0.44
E:C	**Type**	0.33	0.33	0.33	0.17	0.11	0.2
	Cause	0.67	0.67	0.67	0.17	0.11	0
	Process	0	0	0	0.67	0.33	0
	Expectation	0	0	0	0	0.44	0.8

Table 5.1: The C:M and E:C-proportions of the compiler error messages of the tasks.

The proportions of runtime errors follow a similar trend as shown in Table 5.2. The E:C-proportions for the context are similar. Task 5 and 12 are the exception because they have one LoC and one line-number less than the other runtime errors. The E:C-proportion of the Process element is often higher than the compiler error messages, because the first line of the runtime error contain information on the Process: "Traceback (most recent call last)". No other similarities could be found when looking at the E:C proportions of the runtime errors.

Proportions	**Task:**	**5**	**6**	**9**	**10**	**11**	**12**
C:M	**Context**	0.67	0.71	0.64	0.66	0.67	0.71
	Feedback	0.33	0.29	0.36	0.34	0.33	0.29
E:C	**Global source**	0.13	0.1	0.1	0.1	0.1	0.13
	In-code source	0.21	0.21	0.21	0.21	0.21	0.21
	Line of Code	0.67	0.69	0.69	0.69	0.69	0.67
E:C	**Type**	0.17	0.17	0.13	0.13	0.14	0.2
	Cause	0.42	0.42	0.56	0.13	0.07	0.2
	Process	0.42	0.42	0.31	0.66	0.79	0.6
	Expectation	0	0	0	0.07	0	0

Table 5.2: The C:M and E:C-proportions of the runtime error messages of the tasks.

5.1.3 Summary

The Python error messages can be clearly divided in two groups based on structure and component proportions when executed in a Jupyter Notebook.

The compiler error messages uses a context with a line of code, a global source indicator (the file name), and two in-code source indicators which are the line-number of the line of code and a indicator where the error was discovered. The structure of the context of the compiler error message is always the same and is placed above one line of feedback. The feedback of the compiler error messages is always one line that starts with a type of error, but varies in structure and feedback elements after that. The remainder of the feedback contains information either the cause, the process which the compiler was doing, the expectation the compiler was having or a combination of these elements.

The runtime error messages begin and end with a line of feedback with at least 4 lines of context. The first line of feedback starts with the type of error and ends with a description of the process through which the error occurred (*Traceback (most recent call last)*). The context starts with a line of global source, which is the scope the program was in and the filename which contains that scope. It is then followed by three to five line-numbers, each followed by a line of code. The third line starts with a in-code source indicator. The amount of LoC's is dependent on where the code changed scopes because of a function call or the error occurred. If the function call or error is in either the last line or the second to last line of code, then less LoC's will appear in the context. The same is true when the error is in one of the first two lines of code. This pattern of a line of global source with (often) five LoC's repeats for every function call during the execution of the code where the program switches scopes. The last line of feedback has a similar structure to the feedback of the compiler error messages. It starts with the error type and is followed by a variety of possible information such as the cause, the process and the expectation.

In conclusion, the Python error messages as presented by Jupyter Notebook have a focus on Context information, which is different for the compiler and runtime errors based on the way they present the Global Source, In-code Source and LoC('s). The Feedback information of Jupyter Notebook error messages vary greatly in the proportions dedicated to each element of information, but always mention the Type of error and the Cause of the error message.

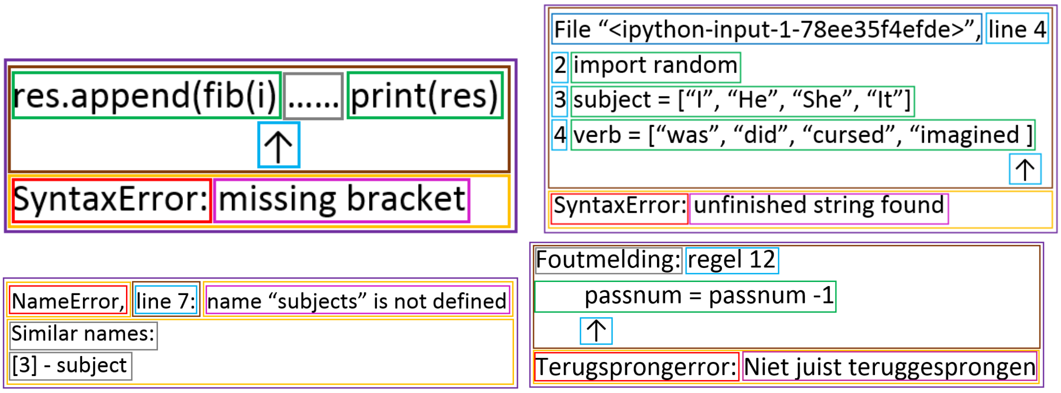
5.2 RQ2: What Information do Novice Programmers expect when encountering Certain Errors?

Each participant has created his or her own error message for each task performed during the experiment. Because participant 1 was unable to create an error message for task 7, this resulted in a total of 76 custom error messages. A few examples of these custom messages are shown in Figure 5.3. All custom error messages are shown in Appendix E.

To investigate the expectations, the custom error message are examined in two ways.

- The content of each custom error message is compared to the original message of that task. The content will be examined on which phrasing was used, if information in the message has changed and in which language it was written.

- The structure of the custom error messages are compared to the original message of that task. This will be done by comparing the structure trees of the custom error message and the original error message.



```
res.append(fib(i)......print(res)
↑
SyntaxError: missing bracket
```

```
File "<ipython-input-1-78ee35f4efde>", line 4
2 import random
3 subject = ["I", "He", "She", "It"]
4 verb = ["was", "did", "cursed", "imagined ]
                                            ↑
SyntaxError: unfinished string found
```

```
NameError, line 7: name "subjects" is not defined
Similar names:
[3] - subject
```

```
Foutmelding: regel 12
    passnum = passnum -1
    ↑
Terugsprongerror: Niet juist teruggesprongen
```

Figure 5.3: Several error messages created by different participants for different tasks

5.2.1 Content Comparing

The custom error messages are examined based on their content and the language used, as this is the part of the error message that the EMCF does not cover. By comparing the content and words used in the custom error messages to the original error message, it is possible to determine if the wording of the original error message satisfies the expectations of the participant. Some of the custom error are written in Dutch. This will only be mentioned when the feedback of the messages is discussed, as the context of the message does not have many words that can be written in Dutch.

Task 1: Missing a Closing Bracket

Task 1 was the only task where the erroneous line reported in the context of the error message was the wrong line of code. Although the error was in line 12, the error message reported that it discovered the syntax error in line 14. All custom error messages for task 1 with context reported line 12 instead of line 14. The only exception is the custom error message of participant 5 (Figure E.28), who reported LoC's 12 *and* 14 in his error message. This may be an indication that the novice programmers do not expect the error message to report the location where the error was discovered, but where the error is. For 2 of the 11 participants the (corrected) context would have been enough to discover the error (Figure E.7 and E.14). Two other participants added the original feedback to the corrected context (Figure E.63 and E.35).

7 participants decided to be more specific in the feedback of their error message, going as far as naming what symbol was missing (a closing bracket). Most of them did not add a lot more words to the feedback, participant 4 and 7 being the exception.

4 custom error messages were written in Dutch. Two of these Dutch error messages contained the Dutch equivalent of the original error message. One of these messages however did add an expected symbol, which indicates that the messages were not translated solely for better understanding.

Task 2: Missing a Comma

The only changes in context content-wise are the in-code source indicator below the LoC, which was changed location most of the time if present. Although this is not a literal change in content, it changes the meaning assigned to it ("here") from "In the number 11" to "Between the number 1 and number 11". Only one custom error message points to the same in-code source, but that error message is a copy of the original.

This same message is also the only message to that has the same content in the feedback. Just like task 1 there are two error messages without feedback. The feedback of the other custom messages is either more specific (4/11), in Dutch (1/11) or both (3/11).

Task 3: Missing a Colon

The change in feedback in the custom error messages are similar to the changes seen in task 1 and 2. 5 messages are more specific, often mentioning what is missing. 3 custom error messages are written in Dutch. One of those messages is a direct translation of the original error message, while the other two are more specific about the missing symbol. The feedback of the last 3 messages are copies of the original error message.

Task 4: an Non-terminated String

The content of the context information in the custom error messages often is equal to, a translation of, or a subset of the context information of the original error message. The only change often seen is that the in-code source indicator under the LoC is moved to point between *"imagined"* and the closing square bracket. Only two custom messages did not follow did this pattern. Participant 6 shortened the LoC. During the interview he explained that he did this for clarity. The feedback mentions a string, but multiple strings are present in the LoC. His version of the LoC uses a filler to shorten the length and hide the "irrelevant" strings in the LoC. The message created by participant 8 has line numbers and two more LoC's than the original message.

There are 3 custom error messages that have a feedback content similar to the original message. However, all these messages expand of the original error message is some way. Participants 1 and 5 reversed the abbreviation *EOL* to *End of Line*, with participant 5 adding more detail about how the EOL caused the error (it was encountered while it was not expected). Participant 7 expended the original message by adding an expectation.

Most of the other custom error messages rephrased the feedback, the message of participant 9 being the exception. Participant 9 only stated the symbol that is expected. The other 7 messages either talk about a (specific) symbol missing (3 custom messages), the string being unfinished before the end of the line (3 custom message), or both (the message of participant 11, see Figure E.73). One of the message that mentions a missing quotation mark is written in Dutch, as is one of the messages talking about an invalid string.

Task 6: Requesting Data from a Non-existing Variable

When comparing the context of the custom error messages to the original error message of task 6, it is obvious that the context (when present) is greatly reduced. A lot of the participants use the structure of the compiler error messages with a single LoC instead of the 5 LoC's that the original message uses. The other content of the context has changed along with it, as most error messages use “Line #” (or its Dutch equivalent in a single case).

The only unique content change in the context is the context in the message created by participant 9. Though the context of his custom error message looks like the original (multiple lines of line-numbers followed by LoC's), it shows a LoC not used in the original error message. This new line is a copy of line 3 of the code, where the variable is defined that the reference should have referenced instead of the non-existing variable. Participant 7 does something similar, although it mentions the similar variable name as part of a list of suggested similar variable names.

The content of the feedback of the custom error messages is similar to the original error message. The participants often just copied the lower feedback of the original error message (*NameError: name “subjects” is not defined*), though some did not copy the type. The few content changes made to the feedback did not change the information in the feedback but the words used to convey the message. Participant 6 used the term *Definitionerror* instead of *NameError*, because he saw the error as a problem with the definition and not with the name of the error. Participant 11 changed the word *name* to *variable*, as he did not understand why the word *name* was used there (see Section 5.3).

Task 7: Not a Correct Amount of Indentation

The content of the context information in the custom error messages often is equal to, a translation of, or a subset of the context information of the original error message. Only two custom messages did not follow did this pattern. Participant 6 added contextual information concerning the amount of indentation used in the erroneous line and the lines before and after. The message created by participant 8 has line numbers and two more LoC's than the original message.

4 of the custom error messages were written in Dutch. Most of these Dutch messages were the same as or similar to the Dutch version of the original error message. One of the Dutch custom messages focuses on the cause of the error by giving the feedback that the unindent is incorrect (see Figure E.12). One other custom error message focused on the cause of the error with similar phrasing (see Figure E.75). All other messages have similarity to the original message when looking at the feedback, though participant 3 used a different phrasing for *any outer indentation level* (see Figure E.19) and participant 7 expended on the expectations (see Figure E.47).

Task 8: No Indentation after an If-statement

Two custom error messages have context content that is not present in either the original message or its Dutch translation. Participant 7 added the LoC's before and after the erroneous LoC with their line numbers to expand the context. Participant 9 expanded the information within the LoC, by visualizing the indentations using symbols.

Participant 9 used this same symbol in the feedback of their message to clarify what the compiler/interpreter expected. This custom message was written in Dutch. This message also specifies that the indentation is expected because of an if-statement, while-loop, or for-loop. The error message of participant 1 also specifies these programming constructions. Participant 8 is more specific about what programming construction expects the indentation (the if-statement). This information was labeled as the process because it gives information of the process the compiler/interpreter was going through during this specific run, instead of general expectations for the previously mentioned programming constructions.

Two messages have feedback content that is different from the original message or its Dutch translation. Participant 7 expands the expectations in his error message by adding a corrected version of the LoC's presented in the context of his message. Participant 11 focused on the cause by stating that the indentation is missing instead of that an indentation was expected.

The other 6 custom message have content similar to the original message. 3 of these message were written in the Dutch language.

Combining Content with Performance

The analysis of each task show a great diversity in the expectations of the participants when confronted with different errors. When combining the analysis with how the participant executed the tasks, the following trends can be found:

- **Context matters:** Task 1 showed an error message where the reported line did not match the erroneous line of code. This displeased some of the participants, two of them saying that they felt betrayed by the error message. In the custom error messages for task 1 the erroneous line is always the reported LoC, indicating that the participants expect the error message to report the line containing the error.

- **Precision of Feedback:** Task 1, 2, and 3 have the same error message feedback, even though their errors are different. Out of the 33 custom error messages created for these three tasks, 22 were more precise about what caused the error. This precision varied from naming the missing symbol to saying that a symbol was missing. The other 11 custom error messages were almost equally distributed over the three tasks (four messages for each of task 2 and 3, three messages for task 1), thus seemingly not favoring a specific error to be solved. This is in contrast with the performance of the tasks, where all participants were able to fix the error of task 3, all but one were able to fix the error of task 2, and only five participants were able to fix the error of task 1. No matter what the error, more precision in feedback than the current messages for a missing closing bracket, comma or colon was expected.

- **Missing a Quotation Mark needs a better Feedback:** The majority of the participants created a custom error message for task 4 where the feedback did not look anything like the feedback of the original message. Most participants stated that they only used the context of the error message to locate, understand and fix the error.

- **More Context is not Better Context:** The error message of task 6, the only runtime error, contained a lot of lines of code. The amount of lines of code was not duplicated by the majority of participants. A few participants indicated that they did not understand why this message had more lines of code in its message, which could be a framing problem caused by the other tasks which only had a single LoC. However, some neglected the context and ended up editing the name of the original variable instead of the "faulty" variable.

- **Similar Names are Relevant Information:** As mention above, some participants fixed the name of the variable definition instead of where the variable name was misspelled. In the error message of participant 7 and 9 both are mentioned, which hints to a feature often found in more complex editors: suggestions of already used but similar variable names.

5.2.2 Structural Comparing

For each task the structure tree of the error message was compared to the structure tree of the custom error messages. Each custom message is compared based on the structure of its categories, its context and its feedback.

To compare the custom error messages to the original message of a task, the same definitions will be used as defined in Section 5.1.1.

The result of the this comparison is shown in Table 5.3. If a participant is not present in the Category column, then the structure of their error message was not the same when looking at category nodes. If a participant is not present in either the Context or Feedback column, then their error message lacked information of that category. Participant 11 is not present in the Context column for task 1 to 4, because his error messages for those tasks did not contain any information about the context of the error. Participant 1 is not present for task 7, because he was enable to produce an error message for that task.

When looking at the similarity in based on the categories, most tasks are structurally the same as around half or more of the custom error messages. The only exception to this is task 6, where none of the participants followed the categorical structure of the original message (feedback, then context, then feedback again). Participants 4, 7, and 11 created error messages that were never structurally the same as the original message based on the category nodes.

When the context of the custom messages are compared to the original message for each task, the participants often prefer a different structure. This could be because the participants often only mention the line-number of the erroneous code, which reduces the context information to two words of in-code source. This may be a unexpected consequence of using a single file of code (with an generated name) with the code always on screen, which would the Global Source and LoC redundant. The only exception to this is task 3, where around half of the participants had a context similar to the original message. Upon closer inspection of the custom error messages, the cause of this exception may be that three of the participants were satisfied with the original error message and copied it without change.

	Category	Context			Feedback		
	Same	Same	Similar	Not Similar	Same	Similar	Not Similar
Task 1	2, 5, 8, 9, 10	2, 3	5, 10	1, 4, 6, 8, 9	2, 4, 5, 6, 7, 8, 10, 11		1, 7, 9
Task 2	1, 5, 8, 9, 10	5, 8, 9, 10	3	1, 2, 4, 6, 7	4, 5, 10, 11	6, 8	1, 7, 9
Task 3	1, 2, 3, 6, 8, 9, 10	2, 3, 6, 8, 9, 10	4,	1, 7	2, 3, 4, 5, 6, 10, 11	8	1, 7, 9
Task 4	1, 2, 8, 9, 10	2, 9, 10	4	1, 3, 6, 7, 8	5	2, 3, 4, 6, 8, 10, 11	1, 7, 9
Task 6			5, 8	1, 2, 3, 4, 6, 7, 9, 11	2, 4, 5, 6, 8, 9, 10		1, 7, 11
Task 7	2, 5, 8, 9, 10	5, 9, 10		2, 4, 6, 7, 8, 11	4, 5, 6, 10	8	2, 3, 7, 9, 11
Task 8	1, 2, 3, 5, 8, 9, 10	2, 3, 5, 8	4, 6, 9, 10	1, 7, 11	2, 3, 4, 5, 10	6	1, 7, 8, 9 11

Table 5.3: Similarity between the message of each task and the messages that each participant created.

Task 6 is the only task where none of the participants had a context structure similar to the original. Task 6 was the only task with a runtime error and the only error message with a runtime error message. A lot of the participants did not understand why there was a different amount of lines of codes, as only one of the five LoC's related to the cause of the error. Some of the participants did have multiple LoC's in their error message, but either only the erroneous line and the two lines above (see Figure E.32 and E.53) or the erroneous line and the other line of code which could be changed to fix the error (Figure E.60). Two of the participants used a context structure more similar to the compiler error message context structure (Figure E.11 and E.18).

The structure of the feedback for most of the tasks is at least similar to at around 50% of the feedback of the custom messages. Task 4 was the task that had less participants that had the same feedback structure than similar feedback feedback structure. Only one of the eleven participants had the same feedback structure as the error message feedback of task 4. Most of the custom error messages for task 4 were the same to the feedback of the first three tasks, whose error messages had feedback consisting of a type and the cause.

Task 6 was analysed differently than the other messages, because it was the only task with an error message with two feedback branches. The structure of the feedback of the custom error message was only compared to the lower feedback. The feedback structure of the messages of the participants for task 6 was almost always the same. The custom error messages of participant 6 expanded the feedback of the original error message by adding information about the expectations, but would have been similar in feedback structure otherwise. Participant 1 and 11 did create error messages with similar feedback structure, as they both did not start their feedback with a type.

5.2.3 Summary

When comparing the expectations of the participants using their custom error messages to the original error message, it seems that there some important differences. The content of the custom error messages show that the precision of the context matters, as all custom error messages created for task 1 pointed to a different line of code than the original error message. The amount of context information should not be "too much", as shown by the custom error messages for task 6. Even though the original error message used five LoC's to describe the context, most participants only used one LoC.

Though there is nothing wrong with the information structure of the feedback of the error messages of task 1, 2, and 3, the participants do expect them to be more precise. A more precise feedback may have prevented that about half of the participants were not able to fix the error in task 1.

Both the content and structure show that the error message for communicating that a String is not terminated does not meet the expectations of the novice programmers. The content of the custom error messages often rephrase the original message in a way that only gives information about the cause.

The structural analyses of the custom error message revealed that not a lot of participants expect the amount of context given to them by the error message of task 6. The error message of task 6 did seem to have the feedback who most closely matched the expectations of the novice programmers.

The result of the content and structural analyses is that the novice programmers expect the context of the error message to be compact but correct. The feedback should be precise enough to understand the exact cause of the error message or what symbol was expected.

5.3 RQ3: Do Error Messages in Python Often encountered by Novice Programmers contain Unknown/Unclear Terms?

The participants highlighted a total of 14 terms which they deemed unclear. Table 5.4 shows these 14 unknown terms (UT), including how many of the participants did not understand these terms. Each term is assigned a code in order to make it easier to reference. There are a few things to note here:

- UT8 is actually a collection of terms, *Traceback (most recent call last)*. These terms were treated as one because the participants did not single out a term and (when deemed unclear) underlined them as a whole.

- Some participants underlined the whole feedback of Task 4 except the type, which would suggest this collection of terms should get the same treatment as UT8. However other participants singled out specific terms within the feedback to be unclear, or did not mark String as an unclear term. The feedback of Task 4 is therefor treated as 4 separate terms.

- Some terms were also part of the type. *Syntax* and *indentation* were part of the type as well as part of other elements of the feedback. When the participant marked these words as unclear, they often only marked the words that was not part of the type.

Code	Term(s)	Present in Task:	# Participants
UT1	Filename	1,2,3,4,6,8	4
UT2	Invalid	1,2,3	3
UT3	Syntax	1,2,3,4	4
UT4	EOL	4	10
UT5	while scanning	4	6
UT6	String	4	3
UT7	literal	4	10
UT8	Traceback (...)	6	3
UT9	name	6	1
UT10	tokenize	7	4
UT11	indentation	7,8	3
UT12	unindent	7	4
UT13	indented	8	3
UT14	block	8	3

Table 5.4: Terms that were marked as unclear by the participants

Some terms were clear outliers. UT9 (*name*) was only marked as unclear by one participant, who explained that it was more the usage of name over variable that was unclear.

Only one of the eleven unclear terms was unique to Jupyter. UT1 is the filename generated by Jupyter Notebook and changes each time the code is run. This unclear term may be unique to the notebook and may not have been present if another editor was used. CSCircles does not mention a file name, as code run in previous executions do not interfere with the current execution. Repl.it lets the programmer code in a named file, which is referenced in the error message when an error occurs.

Some terms do clearly indicate a problem. All terms that were indicated to be unclear by over 50% of the participants are in the same error message. UT4, UT5 and UT7 are all part of Task 4, where the error is caused by a string that is not terminated. EOL (UT4) is the acronym for End Of Line, also know as Newline. It is a special character that is used in text-file storage or console-line printing to indicate that the following characters should be displayed on a new line. This may not be common knowledge among novice programmer, since this knowledge is about low-level representation of data. UT5 and UT7 both concern compiler theory, where *while scanning* concerns the process where the Python code is checked before it is run and *literal* is an term used when making an abstraction of a line of code for lexical analyses. Both terms concern knowledge about how compilers check if the syntax is correct on a more advance level than most novice programmers know.

The error message feedback of Task 4 does not seem targeted towards novice programmers, while the error itself is made quite often by those programmers according to Kohn

[21] and Pritchard [28].

5.3.1 Comparing to their Own Message

That some terms were marked as unclear by an participant did not mean that they did not use it in their own error message. Table 5.5 shows which participants used terms in their own error message that they deemed to be unclear. If an participant only used the Dutch version of the term (UT2, UT3, UT11, UT12, UT13) or used the de-abbreviated form of the term (UT4), then that participant is in bold.

Code	Term(s)	Though unclear, used by participant:
UT1	Filename	6, 9
UT2	Invalid	6, **9**
UT3	Syntax	2, 4, 6, **9**
UT4	EOL	**1, 5, 11**
UT7	literal	1, 5, 7
UT11	indentation	2, **9,** 10
UT12	unindent	**2, 10**
UT13	indented	**2, 4**

Table 5.5: Terms that participants used even though they found them unclear.

Each error message containing an unclear but used term was inspected. This revealed that participant 6 did not actually write down the file name when creating his own error. Participant 6 declared that the error in the code of task 3 could be deduced from the error message and would keep the message as is when asked to create his own error message. He wrote this down as “Kopie” (Dutch for “Copy”). This resulted in the single occurrence of the file name in all error messages of participant 6. If he were to copy the message by hand, he might not have copied the filename.

When looking at the participants that used the (English) term Syntax (UT3) and indentation (UT12), it is revealed that participant 2, 4, and 10 only used these terms as part of the Type. If the participants used these terms in either the Cause or the Expectations of their error messages, then the terms were written in Dutch. A possible cause for the difference in language is the reference material they were given, the Dutch error messages as shown in Appendix C. All elements of Feedback except the Type was translated, because the Type referenced the type of error as documented in the Python documentation (and in Appendix A). The participants might have copied this pattern. Participant 2 has translated the Type once, in their own error message for task 8 (Figure E.13). However, it is unclear why he did so and why only once. Another possible explanation is that these participants do not mind that the Type is based of the English language, as long as the other elements of the feedback are clear.

The term “EOL” (UT4) was de-abbreviated by participant 1, 5, and 11. This de-abbreviation was enough to make the error message more clear for participant 1 and 5.

The remainder of their error message was similar to the original error message, which explains why they still used the term “literal” (UT7). Participant 11 did use the terms “End of Line”, but did change the feedback of his message to focus on the Cause of the error. Participant 7 also used the term “literal” in his error message. This is most likely because he copied the feedback of the error message of task 4 and UT7 was the only term he did not understand, therefor not hindering his debugging process.

The terms “indentation”, “unindent”, and “indented” (UT11, UT12 and UT13 respectively) were, if deemed unclear by the participant, often used in Dutch. These translation were used by most of the participants that deemed the terms unclear. A possible reason why these participants preferred the Dutch translation is because of the way they learned Python. They were taught by a Dutch-speaking teacher who used Dutch words for programming concepts like variables, conditions, indentation, etc. The participants are therefor likely more familiar with the Dutch terms used during the lessons than the English terms used in the error message. This means that the language used during the education of the programmer may influence their comprehension of the error message (and in extension the documentation).

5.3.2 Summary

Python error messages that are often encountered by novice programmers do contain terms that are unknown or unclear, as shown in Table 5.4. The terms that were unclear to over half the participants of the experiment originated from the same error message, the error message of Task 4. Task 4 contained the error of an non-terminated String. The error message of task 4 uses terminology that is not suited for novice programmers, whilst the error is often encountered by this group of programmers.

Some participants still used the terms they deemed unclear, sometimes as part of the type or in their native language. The first may indicate that the participant understood the necessity of the unclear term in the type, the latter may be the result of the language in which they were taught Python/programming. If the lesson about the programming language is not in English but in their native language (like with the participant of this research), it may affect their comprehension of the error messages in a negative way.

5.4 Evaluating the EMCF

This research is the first to use the Error Message Component Framework, which means that means that unforeseen situations could be encountered. This hold especially true for the messages produced by the participants, because they have little to no knowledge on how the compiler processes code or what their limitations are. This section will discuss where the EMCF needed to be adapted or expanded in order to represent the error messages made by the participants.

5.4.1 Parts without Element Classification

When the EMCF was applied to the error messages made by the participants, there were several words that did not fit into any element of information. For example: several messages contained the word “foutmelding” (“error message” in Dutch, Figure E.8 and E.12) or “error” (Figure E.62 and E.76). These words are indicate the presence of an error, but don’t hold information about the location nor the type of error. For this research, “foutmelding” and “error” will be part of the feedback, because these words seem to be taking the place of the Type in the error messages.

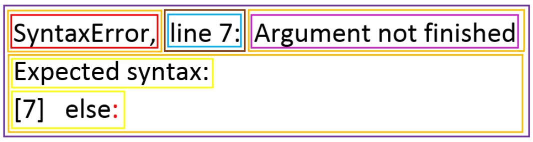
Another error message with parts that did not fit any kind of element of the EMCF is the message created for task 7 by participant 6 (Figure E.40). The last three lines elaborated on the amount of spaces used in the the erroneous lines, as well as the line before and after. These three lines are additional information about where the error occurred thus part of the Context, but are not a Line of Code, in-code source or Global source.

The last outlier is part of the error made for task 6 by participant 7 (Figure E.46). The last two lines of the error message suggests to the programmer which variable they possibly intended. This research classifies these suggestions as part of the feedback, because these elements would be an expectation if the second line of the message was phrased differently (“Expected names” instead of “Similar names”).

To represent the elements above in the Structure Tree, they were indicated as “other information” with the color grey. No type of elements nor categories were added for these parts, because it can not be determined if these parts are indicators of a missing element or category of information, or the result of the unrestricted imagination of the participants. The EMCF needs to be applied to a greater set of error messages (of different programming languages) in order to validate which of these two options is true.

5.4.2 Changing Category within a Single Line

A lot of the error messages made by the participants changed category in a single line. For example: the first (and only) error message line from participant 1 for task one was “Regel 12: Vergeten haakje” which translates to “Line 12: forgotten bracket”. “Line 12” is a clear indication of where the error occurred thus is part of the context. However, “forgotten bracket” is a clear indicator of what caused the error and is part of the feedback. The framework as described in Chapter 3 does not take this situation into account for the structure tree. The error message created by participant 7 for task 3 (shown in Figure 5.4) will be used as an example.



```
SyntaxError, line 7: Argument not finished
Expected syntax:
[7]   else:
```

Figure 5.4: Element distribution in the error message made by participant 7 for task 3

The following steps were taken when a category switch occurs within an error message line: A separate category node is created for the error message line, separating the line from the previous line. The line is then examined from left to right. Each switch of category is documented in the tree with a category node of the new category. When the line ends, the next line is examined. This new line is added as a new category node even if the line started with the same category as the previous line, to clarify that the sequence of categories concerns a single line. This is why “Expected syntax” is started with a new category node in Figure 5.5, even though the previous line started with a category node of the same category.

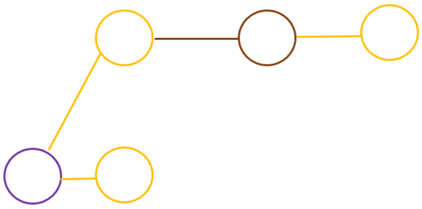


Figure 5.5: The partial Structure Tree of Figure 5.4, based on Categories only

The element nodes of the are added the same way: added left to right, rooted to the category node which contain them. However, the first element node originating from a category node that is part of a multi-category line are placed to the lower-right of the category node, because the right is occupied by other category nodes. This will result in a Structure Tree as seen in Figure 5.6. The quantification of the element, category and message node remain the same,

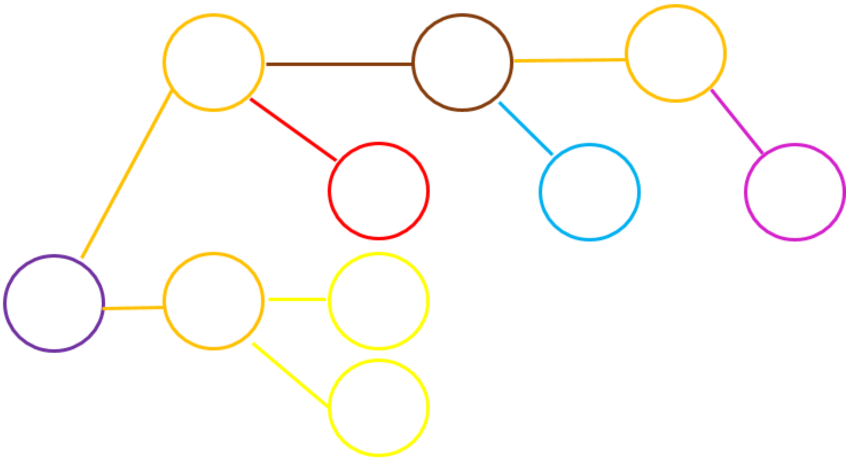


Figure 5.6: The Structure Tree of Figure 5.4 without weight

5.4.3 Changes in the LoC

Some participants made modifications to the erroneous line of code. For example, participant 9 added a symbol to visualise a tab indentation in the LoC (Figure E.62). Participant 7 did something similar in their message for task 8 (Figure E.48), but as a part of the expectations. The amount of information in these (copies of) lines of code is more than the original line of code and can not have the same quantity as any other LoC. These lines are therefor quantified slightly higher, to indicate that information has been added (1 more, equal to the word “tab”).

On the other end were the custom error messages where not all of the line of code was used. Participant 2 only used part of the line of code in two of his error messages (Figure E.7 and E.8). Participant 6 condensed the line of code in one of his error messages, because the feedback mentions the type String and multiple values of that type exist within the erroneous line (Figure E.38). The amount of information in these LoC’s is reduced, thus they should count for a smaller “number of words”. The number has been halved to represent that the line of code was partially copied in the error message.

Chapter 6

Conclusions and Future Work

This thesis originates from the question *Do Python Error Messages match the expectations of novice users?* Eleven novice programmers with a half year of experience with the Python programming language were part of an experiment where they each performed seven tasks. Each task involved fixing a coding error based on its corresponding error message, evaluating the error message and the words used in the message, and creating a custom error message of their own representing how they think the message should be. After analysing the result of the experiment in the previous chapter, this chapter will summarise its conclusions and answer the question central to this thesis.

6.1 Contributions

Methodology: Creating a Custom Error Message

This thesis is the first to use a methodology where participants create their own error message in order to gain a more in-depth view of the information expectations of a user of a programming language. The results, as can be read in Chapter 5, show that this methodology can reveal why some error message do not work as they should. This methodology is another tool to be used in understanding error messages and the users who interact with them.

The Error Message Component Framework

This thesis presents the Error Message Component Framework (EMCF), a framework which can be used to determine which components of information are present in an error message, what proportion of the error message is dedicated to a specific component and how the error message is structured. The EMCF divides information in error messages into Context and Feedback, describing where and what the error is respectively. Identifiable elements of the Context are the Global Source (where is the error on a file/scope level), In-code Source (where is the error in the code), and Line of Code (copies of lines of code). Feedback can consist of the elements Type (what Type of error was encountered), Cause (what caused the error), Process (what was the compiler/interpreter/runtime-environment doing or attempting to do), and Expectation (what was expected during the process).

The EMCF enabled the analyses of the original Python error messages and the custom error message as created by novice programmers. The full description of the EMCF is written in Chapter 3.

6.2 Conclusions

This thesis began with the question **RQ**: *Do Python Error Messages match the expectations of novice users?* Three sub-questions were created based on this question:

> **RQ1**: What components do Python error messages consist of?
>
> **RQ2**: What information do novice programmers expect when encountering certain errors?
>
> **RQ3**: Do error messages in Python often encountered by novice programmers contain unknown/unclear terms?

The Python error messages created by the Jupyter Notebook were analysed using the EMCF to answer **RQ1**. This analyses revealed that the components used in the error messages depended on the kind of error that was made. Compiler errors created error message that consist of three lines of Context whose component structure is always the same, followed by a single line of Feedback that starts with a type. The structure of the feedback is variable after the type, though it seems that it often starts with the Cause. Runtime errors are shown differently. Runtime error messages start with a single line of Feedback, which always show the Type of error followed by Process (*"Traceback (most recent call last)"*). The Feedback is followed by a variable amount of Context. This context always start with a Global Source (the filename and the scope), followed by a variable amount of line-numbers (In-code Source) and LoC's. The runtime error message ends with a line of Feedback, whose structure is similar to the Feedback of the compiler error message.

The answer to **RQ2** was revealed by an analyses of the custom error message, the error messages created by novice programmers. Novice programmers want the Context component to be compact but precise. The custom error messages revealed that the novice programmers might not expect a Global Source or LoC in the error message when the code is in a single file with a small amount of code. The preciseness-expectation of the context was derived from the custom error messages for Task 1, where the original error message copied a different LoC than the erroneous line. All participants "corrected" this in their custom error message. The custom error messages revealed that novice programmers often want the feedback to be precise enough to understand the exact cause of the error. Custom error messages created for reporting the errors in Task 1, 2, and 3 were often more precise about what caused the error or what symbol was expected. The amount of detail in the Feedback of the custom error messages varied, as some reported that *a* symbol was missing while another mentioned the exact symbol that was missing.

The analyses done for **RQ2** showed some differences in the structure of the original error messages and the expectations of the novice programmers. Though the structure of the custom error message was often the same or at least similar, there are two error messages that stand out. The structure of the Context of task 6, only runtime error message presented to the novice programmers, was not the same as any of the custom error messages and only similar to a few of them. This means that no participant copied the Context into their own error message. The participants often opted for a Context structure similar to the compiler error message context structure, which only consisted of a single LoC instead of five. This could be a framing issue, as this was the only runtime error message. However, some participants remarked that they skipped the context entirely and started searching in the code, which is an indication that adding more information is not always helpful.

The other error message that stood out was the error message of Task 4, which contained a non-terminated String. The feedback of the original message was often similar to that of the custom error messages, as they both started with a Type component followed by a Cause component. This is where the similarities end. Very few of the participants copied the content of the Feedback of task 4. The cause which can be found in the answer to **RQ3**. Each participant indicated which words or terms they did not understand or though were unclear. The two term that were deemed the most unclear, both were not understood by 10 out of the 11 participants, were both part of the same error message: the error message of task 4. The terms *EOL* and *literal* are both terms that can not be expected to be understood by programmers who just started creating their programs, unless specifically taught. When looking for the terms the participants found unclear in their own custom error messages, it appeared that some participants still used the terms even though they did not understand them. One of the possible causes was that the terms were only used in the Type component, meaning that the participant did not understand the term but may have understood its necessity in the Type. The other cause was that the term was used in their native language, as they may be able to understand the translated version of the term.

With the three sub-questions answered, the focus is back on the question it began with: **RQ**: *Do Python Error Messages match the expectations of novice users?* As has become evident from the latter two sub-questions, there are strong indicators that the Python error messages (as presented by Jupyter Notebook) do not match the expectations of novice users. The context of the runtime error messages of Jupyter Notebook may contain too much information to be easily understood by novice programmers. There are also indicators that novice programmers do not expect certain context components like global source or LoC's when they are working with Python programs consisting of less than twenty lines of code, though this may be a result of the experimental setup. When it comes to the precision of reporting the context, the novice programmers expect that the error message reports where the error is, not where the compiler/interpreter/runtime-environment discovered the error. Though the novice programmers seem happy with the component structure of the feedback of the error messages, they expected more precision in the content of the feedback. There was no consensus about how much precision was enough, as the participants differed in expectations. Based on the expectations of the novice programmers and the terms that were unclear to them, it can be concluded that the feedback of the error message displayed when the program contains a non-terminated String is not aimed towards novice programmers.

This error message has to be changed, as the mistake it is reporting is a minor mistake often made by novice programmers[21].

6.3 Discussion

6.3.1 Eye Tracking and Why its Data was not Used

The setup of the experiment of this thesis was created with eye tracking in mind. It influenced the editor that would be used, as the software used by the eye tracker could open the Jupyter Notebook with the error message already present. The goal of using an eye tracker was to precisely pinpoint which parts of the error messages were used, which would reveal how each component or element of information from the EMCF was used during the debugging process. The software that comes with the eye tracker is able to record how much time the participant gazes in an Area of Interest (AoI). AoI's were created for each component of an error message, which would reveal how much time the participant would spend reading the different components of the error message.

Though eye tracking data was recorded during the execution of the experiment, the data was not used to answer any questions. The eye tracking data was accurate enough to interpret what was read, but not accurate enough to measure how long the participant took to read components of the error message. The eye tracker was calibrated with a 9-point calibration at the start of the experiment. This did not prevent the data from being inaccurate.

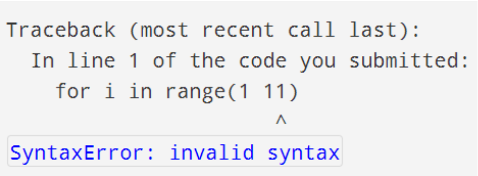
The eye tracking data was influenced by a human factor. The software that comes with the eye tracker enables its user to offset the data by a certain amount of pixel in order for the data to be more representative of the gaze of the user. Not doing this would sometimes result in loads of gaze points in empty parts of the screen. Because the AoI's are very small in height (each line is only 2.8% of the height of the screen), even a small change in the vertical offset can change how much time the used looked at the AoI according to the data.

In addition, some abnormalities in the gaze data could not be fixed with changing the offset. Several times in the data it is clear that the participant is reading a sentence in the error message from left to right. The gaze data shows that some participants slightly elevates their fixation on screen as their gaze progresses to the right. This would result in a time reading where the gaze begins under the sentence and end on the sentence, or where the gaze begins on the sentence and ends above the sentence. In both cases the data was not representative for how long the participant looked at that sentence. Another abnormality was that in some cases the gaze data corresponding with reading the error message could not be overlapped with the actual error message. Because the error messages did not have any readable information above its first sentence and below its last sentence, it was easy to deduce which gaze points were related to the participant reading the error message. Mapping these gaze points to the error message within the software was sometimes not possible, because the error message was sometimes bigger or smaller than the area covered by error-message-related gaze points.

6.3.2 Disadvantages of the Chosen Editor

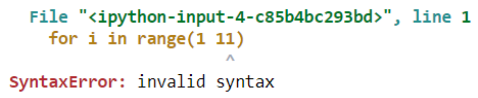
The Jupyter Notebook was chosen as the editor for the experiment of this thesis. The usage of Jupyter Notebook did have some downsides, the first of which was that the error messages of the notebook for syntax and indentation errors are not the same as the error messages displayed by CSCircles. Figure 6.1, 6.2, 6.3, and 6.4 show the error message from different Python editors, among which Jupyter Notebook and CSCircles, when they encounter the same error in the same code.

Although the error messages of Jupyter Notebook do not correspond exactly with the error messages from the CSCircles, they are more similar to the CSCircles error messages than the error messages generated by create.withcode.uk in Figure 6.4. Although the messages from Jupyter Notebook and Repl.it (Figure 6.3) are similar, Repl.it has in-code error highlighting. Because CSCircles does not have any in-code error highlighting, Jupyter Notebook was chosen as the editor over other Python editors.



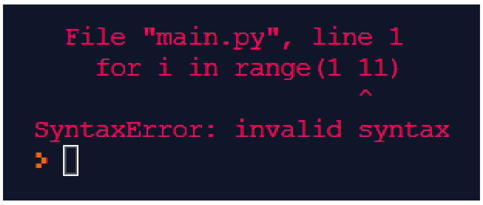
```
Traceback (most recent call last):
  In line 1 of the code you submitted:
    for i in range(1 11)
                     ^
SyntaxError: invalid syntax
```

Figure 6.1: Error message created with CS Circles



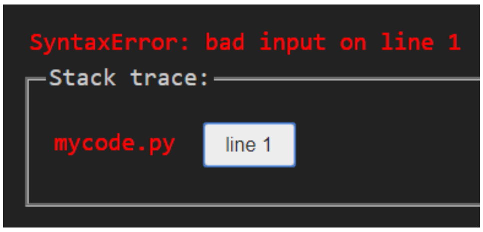
```
  File "<ipython-input-4-c85b4bc293bd>", line 1
    for i in range(1 11)
                     ^
SyntaxError: invalid syntax
```

Figure 6.2: Error message created with Jupyter Notebook



```
  File "main.py", line 1
    for i in range(1 11)
                     ^
SyntaxError: invalid syntax
```

Figure 6.3: Error message created with Repl.it



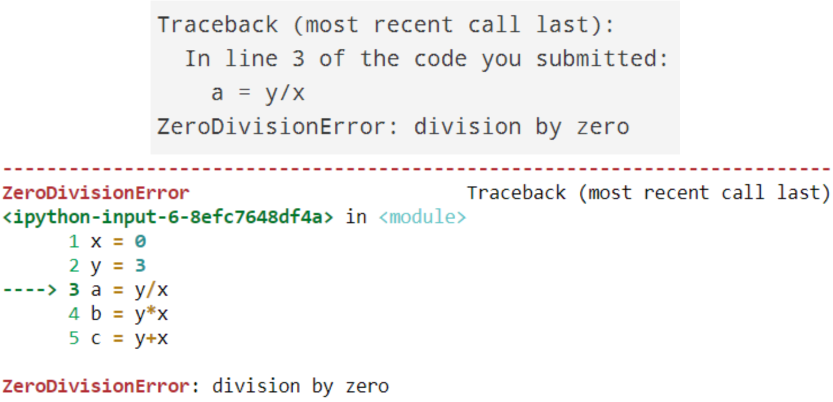
```
SyntaxError: bad input on line 1
Stack trace:
  mycode.py   line 1
```

Figure 6.4: Error message created with create.withcode.uk

Another downside is the way runtime errors are presented. Jupyter Notebook presents these kind of errors with messages with three to five lines of code (depending on where the occurs). CSCircles presents these kind of errors with only one line of code, the line where the error occurs. This difference is shown in Figure 6.5.

6.3.3 The Weight in the Structure Tree

The EMCF uses a structure tree to represent what information each component of the error message contains. These components are represented as nodes in the tree, where each node has the weight equal to the number of words in that component.



```
Traceback (most recent call last):
  In line 3 of the code you submitted:
    a = y/x
ZeroDivisionError: division by zero
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-8efc7648df4a> in <module>
      1 x = 0
      2 y = 3
----> 3 a = y/x
      4 b = y*x
      5 c = y+x

ZeroDivisionError: division by zero
```

Figure 6.5: Difference between a runtime error in CS Circles and Jupyter notebook

Although this gives a general idea on what information the error messages focuses, this method of adding weight is not perfect. The first problem with just counting the number of words in a message is that it is language dependant. For example: “Error message” is two words, while its Dutch equivalent “Foutmelding” is one word. The second problem is abbreviations, like the abbreviation “EOL” in task 4. “EOL” was counted as one word, as it is a special character and characters were counted as one word. Some participants reversed the abbreviation of “EOL” to “End of Line” in their custom error message, which was counted as three words. Their meaning has not changed, but the amount of words has.

A more complex method of assigning weight to nodes in the structure tree is needed to represent the full information distribution of the error message. The current method of counting words was sufficient to compare the proportion of components between the original error messages, as they were all written in English and it could be assumed that words have the same amount of information. Because some of the custom error messages were written in Dutch and each message for the same error was written by a different person, it was deemed unreliable to use the amount of words as a measure for amount of information.

6.3.4 How to Make Error Messages Understandable

This research shows that there are some Python error messages that contain jargon or other terms that were deemed unclear by the participants. The acronym EOL was only known to one participant, who did not understand why the acronym was there until he read the Dutch translation. Although the are several ways to address this problem, each of the solutions has their own hurdle.

The first possible solution is to create a custom compiler/interpreter or programming environment, where the error messages are designed for novice programmers. This would be a way to address the problem directly in a way that could be beneficial to both beginning programmers and teachers. However, this could create a gap between the Python official releases and the novice programmer. When the programmer want to switch to an other programming environment where the official Python release is used, they might encounter the same errors but at a different moment. It is possible that the programmer is knowledgeable enough about programming and Python to resolve the errors, but is also possible that they want to stick to the custom programming environment because the feedback is familiar. Programming environments that are designed for novice programmers often are limited in their capabilities. CSCircles[29] for example only works for programs with a single file, since its programming environment does not have to option to work in multiple files. CSCircles is also limited in the usage of libraries.

Another solution would be to change the standard. If the participants of this research have a hard time understanding some error messages or resolving some errors, it is not unlikely that other novice programmers would encounter the same problems. Changing the content of the error messages most likely This kind of change can not be taken likely though. If the standard would be updated in a way that would affect everyone using it, the changes would have to be tested and researched. What changes would improve the understanding of the error? How would the error messages be tested? How will the existing community respond to these changes? How much tutorials, courses and forums need updating or will become legacy? A change of the Python standard may be good for the long run, but it will take a long time for it to be good.

A last solution would be education. This would require no change to the compiler/ interpreter, but does require time. Whenever a concept of the Python programming language might introduce new types of errors, then the student should be educated on what this error message means. This does add to the cognitive load when learning a new concept, which means that the error message, what it means and how to fix the error may not stick with the student as it is information that is only used occasionally. Because it requires the least amount of change in the implementation of the language and requires no transition from a beginner programming environment to the official release, it seems that education is the most feasible solution for preparing novice programmers to understand the messages about error they will eventually make.

6.3.5 Error Messages in the Native Language

The participants were presented Dutch versions of the error message for each task to determine if they would understand the message that was written if it was translated in their native language. The Dutch versions of the error messages were created for this experiment and are not official translations. These translated error messages resulted in a few participants creating Dutch error messages, which was not encouraged but also not discouraged. Some of these Dutch error messages were translations of the original error message. This could implicate that the words used might not always be the problem when trying to understand the error message. The language in which it is written also matters.

Localisation of the programming language and feedback is not a new topic of research. Dasgupta and Hill [7] determined that novice Scratch programmers who used the localised language and interface demonstrate new programming concepts at a faster rate than users from the same countries whose interface was in English. A study done by Feijóo-García et al. confirms this[11]. Based on an experiment with English-speakers and Spanish-speakers who both had to use the English interface of Scratch to solve a problem, their findings suggest that the native language affects how the participant interacts with the interface, though it did not hinder the participants ability to create a proper solution.

These two papers concern the block-language Scratch, but similar barriers are present when a non-native English speaker is faced with a textual programming language. Guo [14] used a survey to determine which barriers native and non-native English speakers faced when learning programming. These barriers concerned programming itself (problems with reading and writing code) as well as other elements of programming (problems with reading instructional materials, technical communication, and learning English/programming at the same time).

None of these papers investigate if or how localised error messages would help. Guo [14] has several responses that it would not work, because all documentation is in written in English and searching for localised terms is less effective than searching the original, English terms. These are valid arguments, as the help center of StackOverflow (a forum for asking developing-related questions) has “Correct use of English spelling and grammar to the best of your ability” as one of its guidelines for getting your question to be accepted by their server[1]. The error messages created by participant 4, 9 and 10 show that there might be support for localising error message, at least in the initial stages of learning to program. Though it is unlikely to change the general consensus that English is the base language of programming languages, it should be possible to support novice programmers with error messages in a language they understand at least until they are more familiar with the language.

6.3.6 Layered Feedback

Different (novice) programmers have different information needs concerning error messages. A way to facilitate this difference in need is by incorporating the extensible help as suggested by Traver[36]: A short message by default, a brief explanation or example if the short message is not enough, and lastly suggestions on how to solve the error. A layered approach based on the elements of feedback in the EMCF could also be possible. For example: The initial message gives information on the Type and the Source. If that is not enough information, then the Process will be added to the feedback. Lastly an Expectation is added to the message.

6.3.7 Changing the Role of the Compiler

The method of information layering as described in section 6.3.6 would be able to satisfy different information needs when making a mistake. A novice programmer however may still be learning how to program and the error may be the result of an misconception or a gap in knowledge. An extra layer may be needed to address this group.

Kirschner [20] distinguishes three ways to give feedback to students who made a mistake. The first way of giving feedback is Corrective feedback, where the feedback tells if a mistake was made and if so, what it should have been. Kirschner calls this *single loop* feedback, which focuses on the action of the student and the visible result (a correct or incorrect

[1] https://stackoverflow.com/help/quality-standards-error

result). An more in-depth way of providing feedback is giving Directive feedback. Directive feedback tells the student what the error is and how to fix it. This *double loop* feedback focuses on how the students performed en how he/she can improve. The most effective way to give feedback when looking at learning is Epistemic feedback or *triple loop* feedback. This method of feedback uses questions to direct the students towards the error and to help the students gain a deeper understanding of why the current solution is not correct. Because the student is more engaged with thinking about and correcting the error, it is more likely that the student has learned from his/her mistake.

The feedback of the error messages in the tasks of this research can not be easily categorised as single loop, double loop, or triple loop feedback. The messages lack the "correct" solution required for it to be single loop feedback, do not steer towards a correct solution which is part of the double loop feedback, and does ask guiding questions which would make it part of triple loop feedback. When looking at the ways that error is addressed, the feedback of the error messages most closely resembles Directive feedback (double loop). The feedback of the error messages tries to tell the programmer what is wrong instead of only saying that the code is incorrect.

If the role of the compiler would change from a translator to that of an educator, it should give Epistemic feedback to help the novice programmer learn the language. Giving proper epistemic feedback is a lot harder to do than giving directive feedback, as it involves a lot more insight into the abilities of the programmer who made the mistake. The compiler has to know somehow if the user is able to firstly understand the question asked (thus it needs to use words that the user understands) and secondly answer the question (does the user know how to fix error based on the question and their current knowledge).

A recently published programming language Hedy[16] could be a possible adapter of giving epistemic feedback. The Hedy Programming language is a *gradual* language with different levels of language complexity. Hedy starts as a language similar to natural language and over several levels increases its complexity until the language is similar to the Python programming language. The benefit of this language when attempting to implement epistemic feedback would be that each level will not expect the user to be familiar with higher level knowledge. Each level can also expect the user to be familiar with lower lever knowledge. The combination of these limits and prior knowledge can be the basis for creating epistemic error messages which can be both understood and resolved by the users when they are at a certain level of Hedy.

6.4 Future work

6.4.1 Validating the EMCF

The Error Message Construction model as presented in Chapter 3 has been created for this experiment to compare error message structure between the Python error messages and the error messages created by the participants. This model has not been tested in other contexts, such as other programming languages.

This model also needs to be validated for other programmers. If an expert programmer is presented with the same errors and the same component definition, will they end up with the same deconstruction of the error messages? Will the components be sufficient to cover every element of the error message or will the programmer miss a component? A large-scale survey among programmers is needed to evaluate the usability and validity of the model.

6.4.2 Looking at Runtime Errors

After the pilot the decision was made to reduce the amount of tasks to fit the intended timeframe. This excluded all but one runtime error. The runtime that was included (task 6) was perceived as "having a lot of unnecessary context/information". This opinion may be biased, because it is the only error message with five lines of code instead of one. This one of the downsides of using Jupyter Notebook as the editor, because it seems to be unique how it presents runtime errors.

This experiment can be repeated with different tasks to investigate this bias. By using runtime errors only (task 5, 6, 9, 10, 11, and 12) the anchor of using a single line of code in the error message is avoided. This setup would still allow for investigation of the effectiveness of the error messages and the possible information overload.

6.4.3 Looking at Different Python Editors

This research used a locally hosted Jupyter Notebook to investigate the effectiveness of the error message. One of the two main reasons for this choice was because of its similarity with the programming editor used by the participant while they were learning Python.

Python editors do not always have the same error message for the same error in the same code, as shown in Section 6.3.2. This is especially true when the editors with enhanced error messages are included.

The participants who created the messages in Chapter E used the error messages presented to them (as seen in Chapter B and C) as the basis for their own messages. Reproducing this thesis with a different editor could therefor result in different custom error messages. This reproduction study could also verify the impact of the reference error message on the custom error messages.

6.4.4 Creating and Validating Error Messages based on the EMCF

The EMCF has been used to identify different kinds of information in existing error messages and to deconstruct the error messages created by the participants. This framework can also be used to create new error messages with a different distribution and structure of type of information.

For example: What kind of error messages are more effective, messages with minimal context or messages with minimal feedback? Do programmers use error messages differently if the feedback is above the context instead of the other way around? What happens if the feedback only contained one element of information?

Answering these question would improve the understanding of what kind of error message the user wants from the compiler/interpreter and what kind of structure it should have.

Bibliography

[1] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. Do developers read compiler error messages? In *Proceedings of the 39th International Conference on Software Engineering*, pages 575–585. IEEE Press, 2017.

[2] Titus Barik et al. Error messages as rational reconstructions. 2018.

[3] Brett A Becker. An exploration of the effects of enhanced compiler error messages for computer programming novices. 2015.

[4] Brett A Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 126–131. ACM, 2016.

[5] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 177–210. 2019.

[6] Pierre Carbonnelle. Pypl popularity of programming language index. `https://pypl.github.io/PYPL.html`. (Accessed on 03/01/2019).

[7] Sayamindu Dasgupta and Benjamin Mako Hill. Learning to code in localized programming languages. In *Proceedings of the fourth (2017) ACM conference on learning@ scale*, pages 33–39, 2017.

[8] Morris Dean. How a computer should talk to people. *IBM Systems Journal*, 21(4): 424–453, 1982.

[9] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 273–278. ACM, 2014.

[10] Paul Denny, James Prather, and Brett A Becker. Error message readability and novice debugging performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 480–486, 2020.

[11] Pedro Guillermo Feijóo-García, Keith McNamara Jr, and Jacob Stuart. The effects of native language on block-based programming introduction: A work in progress with hispanic population. 2020.

[12] Python Software Foundation. Built-in exceptions python 3.7.2 documentation. `https://docs.python.org/3/library/exceptions.html`. (Accessed on 02/14/2019).

[13] Diana Franklin, Phillip Conrad, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, Gerardo Aldana, Paulo Almeida-Tanaka, Brynn Kiefer, et al. Assessment of computer science learning in a scratch-based outreach program. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 371–376. ACM, 2013.

[14] Philip J Guo. Non-native english speakers learning computer programming: Barriers, desires, and design opportunities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–14, 2018.

[15] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1019–1028. ACM, 2010.

[16] Felienne Hermans. Hedy: A gradual language for programming education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 259–270, 2020.

[17] Felienne Hermans and Efthimia Aivaloglou. Teaching software engineering principles to k-12 students: a mooc on scratch. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, pages 13–22. IEEE, 2017.

[18] James J Horning. What the compiler should tell the user. In *Compiler Construction*, pages 525–548. Springer, 1974.

[19] Ambikesh Jayal, Stasha Lauria, Allan Tucker, and Stephen Swift. Python for teaching introductory programming: A quantitative evaluation. *Innovation in Teaching and Learning in Information and Computer Sciences*, 10(1):86–90, 2011.

[20] PA Kirschner, L Claessens, and S Raaijmakers. Op de schouders van reuzen. inspirerende inzichten uit de cognitieve psychologie voor leerkrachten.[on the shoulders of giants. inspiring insights from cognitive psychology for teachers], 2018.

[21] Tobias Kohn. The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 524–530. ACM, 2019.

[22] Tobias Kohn and Bill Manaris. Tell me what's wrong: A python ide with error messages. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1054–1060, 2020.

[23] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Learning computer science concepts with scratch. *Computer Science Education*, 23(3):239–264, 2013.

[24] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? *ACM SIGCSE Bulletin*, 40(1):168–172, 2008.

[25] JC Olabe, MA Olabe, X Basogain, and C Castaño. Programming and robotics with scratch in primary education. *Education in a technological world: communicating current and emerging research and technological efforts*, pages 356–363, 2011.

[26] Raymond S Pettit, John Homer, and Roger Gee. Do enhanced compiler error messages help students?: Results inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 465–470. ACM, 2017.

[27] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 acm conference on international computing education research*, pages 74–82, 2017.

[28] David Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 1–8. ACM, 2015.

[29] David Pritchard and Troy Vasiga. Cs circles: an in-browser python course for beginners. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 591–596. ACM, 2013.

[30] Brian Randell. Colossus: Godfather of the computer. In *The Origins of Digital Computers*, pages 349–354. Springer, 1982.

[31] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S Silver, Brian Silverman, et al. Scratch: Programming for all. *Commun. Acm*, 52(11):60–67, 2009.

[32] Jeff Rufinus and Yana Kortsarts. Teaching an introductory programming course for non-majors using python. *Director*, page 07, 2006.

[33] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using scratch in five schools. *Computers & Education*, 97: 129–141, 2016.

[34] Ben Shneiderman. Designing computer system messages. *Communications of the ACM*, 25(9):610–611, 1982.

[35] TIOBE. Python — tiobe - the software quality company. `https://www.tiobe.com/tiobe-index/python/`. (Accessed on 03/01/2019).

[36] V Javier Traver. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 2010.

[37] Jacques Wainer and Eduardo C Xavier. A controlled experiment on python vs c for an introductory programming course: Students outcomes. *ACM Transactions on Computing Education (TOCE)*, 18(3):1–16, 2018.

Appendix A

Python Errors

These are nearly all error types that can be produced by Python. The error messages missing are all subclasses of OSError[12].

- **AssertionError**
  Raised when an assert statement fails.

- **AttributeError**
  Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)

- **EOFError**
  Raised when the input() function hits an end-of-file condition (EOF) without reading any data. (N.B.: the io.IOBase.read() and io.IOBase.readline() methods return an empty string when they hit EOF.)

- **FloatingPointError**
  Not currently used.

- **GeneratorExit**
  Raised when a generator or coroutine is closed; see generator.close() and coroutine.close(). It directly inherits from BaseException instead of Exception since it is technically not an error.

- **ImportError**
  Raised when the import statement has troubles trying to load a module. Also raised when the from list in from ... import has a name that cannot be found.

  The name and path attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the module that was attempted to be imported and the path to any file which triggered the exception, respectively.

- **ModuleNotFoundError**
  A subclass of ImportError which is raised by import when a module could not be located. It is also raised when None is found in sys.modules.

- **IndexError**
  Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, TypeError is raised.)

- **KeyError**
  Raised when a mapping (dictionary) key is not found in the set of existing keys.

- **KeyboardInterrupt**
  Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly. The exception inherits from BaseException so as to not be accidentally caught by code that catches Exception and thus prevent the interpreter from exiting.

- **MemoryError**
  Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (Cs malloc() function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

- **NameError**
  Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

- **NotImplementedError**
  This exception is derived from RuntimeError. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

- **OSError**
  This exception is raised when a system function returns a system-related error, including I/O failures such as file not found or disk full (not for illegal argument types or other incidental errors).

- **OverflowError**
  Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise MemoryError than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating point exception handling in C, most floating point operations are not checked.

- **RecursionError**
  New in version 3.5. This exception is derived from RuntimeError. It is raised when

the interpreter detects that the maximum recursion depth (see sys.getrecursionlimit()) is exceeded.

- **ReferenceError**
  This exception is raised when a weak reference proxy, created by the weakref.proxy() function, is used to access an attribute of the referent after it has been garbage collected.

- **RuntimeError**
  Raised when an error is detected that doesnt fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

- **StopIteration**
  Raised by built-in function next() and an iterator's __next__() method to signal that there are no further items produced by the iterator.

  The exception object has a single attribute value, which is given as an argument when constructing the exception, and defaults to None.

  When a generator or coroutine function returns, a new StopIteration instance is raised, and the value returned by the function is used as the value parameter to the constructor of the exception.

  If a generator code directly or indirectly raises StopIteration, it is converted into a RuntimeError (retaining the StopIteration as the new exceptions cause).

- **StopAsyncIteration**
  Must be raised by __anext__() method of an asynchronous iterator object to stop the iteration.

- **SyntaxError**
  Raised when the parser encounters a syntax error. This may occur in an import statement, in a call to the built-in functions exec() or eval(), or when reading the initial script or standard input (also interactively).

  Instances of this class have attributes filename, lineno, offset and text for easier access to the details. str() of the exception instance returns only the message.

- **IndentationError**
  Base class for syntax errors related to incorrect indentation. This is a subclass of SyntaxError.

- **TabError**
  Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of IndentationError.

- **SystemError**
  Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

- **SystemExit**
  This exception is raised by the sys.exit() function. It inherits from BaseException instead of Exception so that it is not accidentally caught by code that catches Exception. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to sys.exit(). If the value is an integer, it specifies the system exit status (passed to Cs exit() function); if it is None, the exit status is zero; if it has another type (such as a string), the objects value is printed and the exit status is one.

  A call to sys.exit() is translated into an exception so that clean-up handlers (finally clauses of try statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The os._exit() function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to os.fork()).

- **TypeError**
  Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

  This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, NotImplementedError is the proper exception to raise.

  Passing arguments of the wrong type (e.g. passing a list when an int is expected) should result in a TypeError, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a ValueError.

- **UnboundLocalError**
  Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of NameError.

- **UnicodeError**
  Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of ValueError.

- **UnicodeEncodeError**
  Raised when a Unicode-related error occurs during encoding. It is a subclass of UnicodeError.

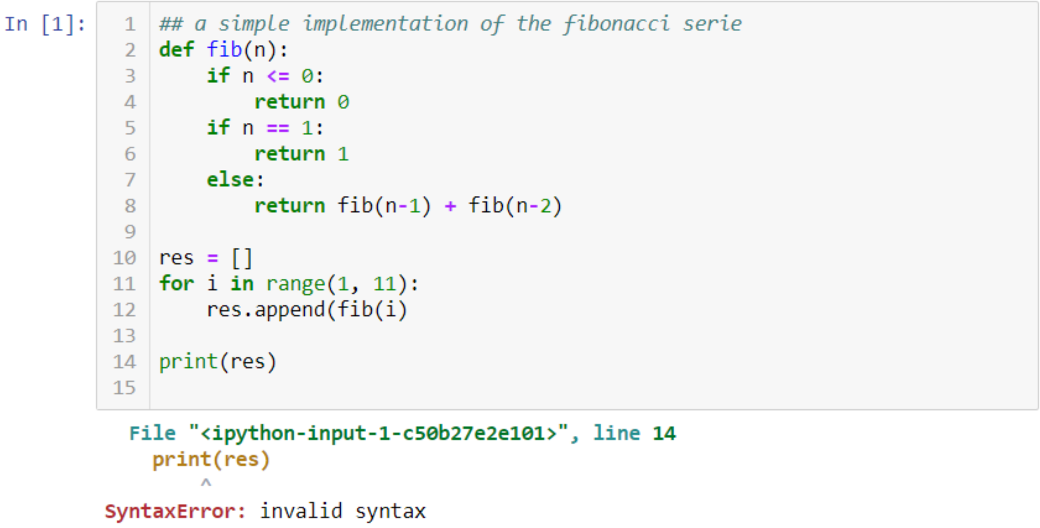
- **UnicodeDecodeError**
  Raised when a Unicode-related error occurs during decoding. It is a subclass of UnicodeError.

- **UnicodeTranslateError**
  Raised when a Unicode-related error occurs during translating. It is a subclass of UnicodeError.

- **ValueError**
  Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.
- **ZeroDivisionError**
  Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

Appendix B

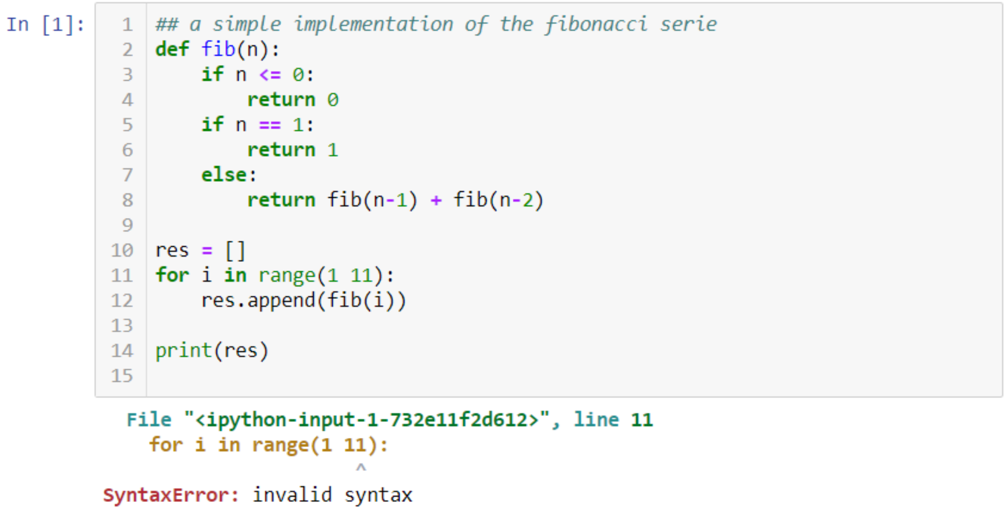
Tasks



```python
## a simple implementation of the fibonacci serie
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

res = []
for i in range(1, 11):
    res.append(fib(i)

print(res)
```

```
  File "<ipython-input-1-c50b27e2e101>", line 14
    print(res)
        ^
SyntaxError: invalid syntax
```

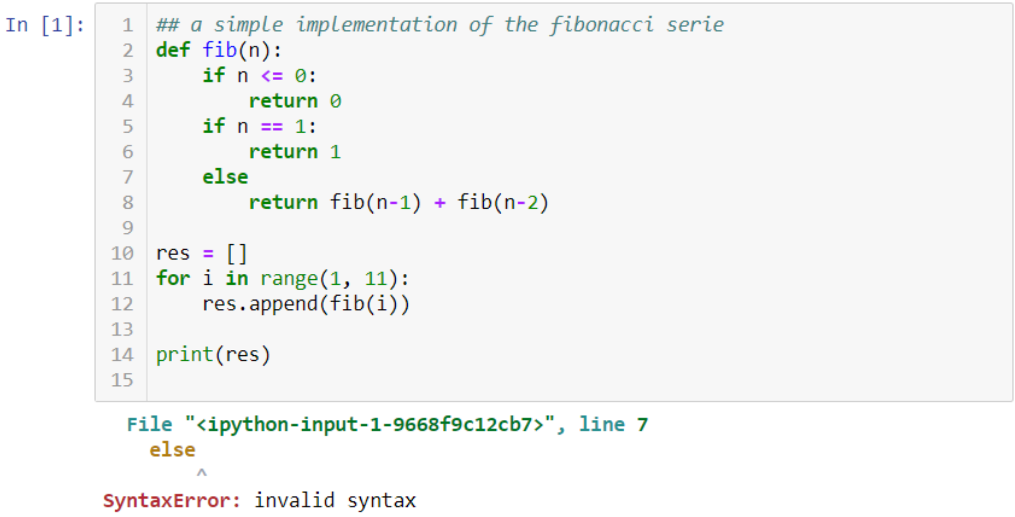
Figure B.1: Task 1, a missing closing bracket



```
In [1]:
## a simple implementation of the fibonacci serie
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

res = []
for i in range(1 11):
    res.append(fib(i))

print(res)

```

```
  File "<ipython-input-1-732e11f2d612>", line 11
    for i in range(1 11):
                      ^
SyntaxError: invalid syntax
```

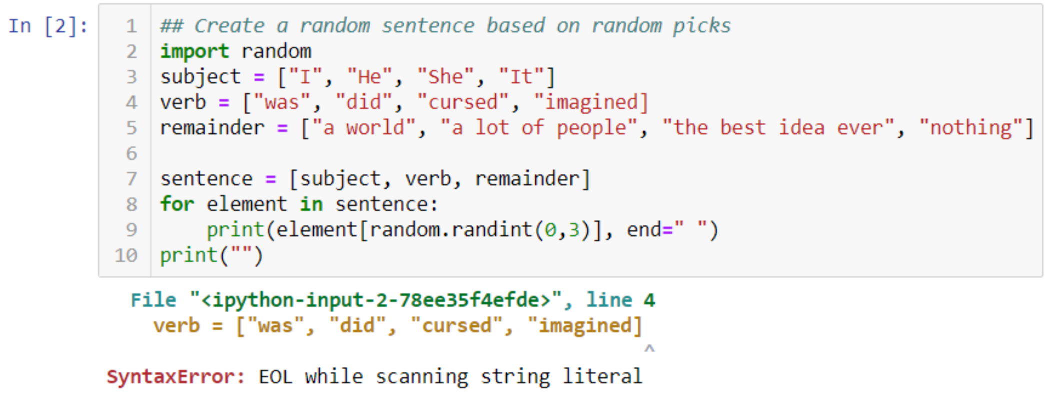
Figure B.2: Task 2, a missing comma



```
In [1]:
## a simple implementation of the fibonacci serie
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else
        return fib(n-1) + fib(n-2)

res = []
for i in range(1, 11):
    res.append(fib(i))

print(res)

```

```
  File "<ipython-input-1-9668f9c12cb7>", line 7
    else
        ^
SyntaxError: invalid syntax
```

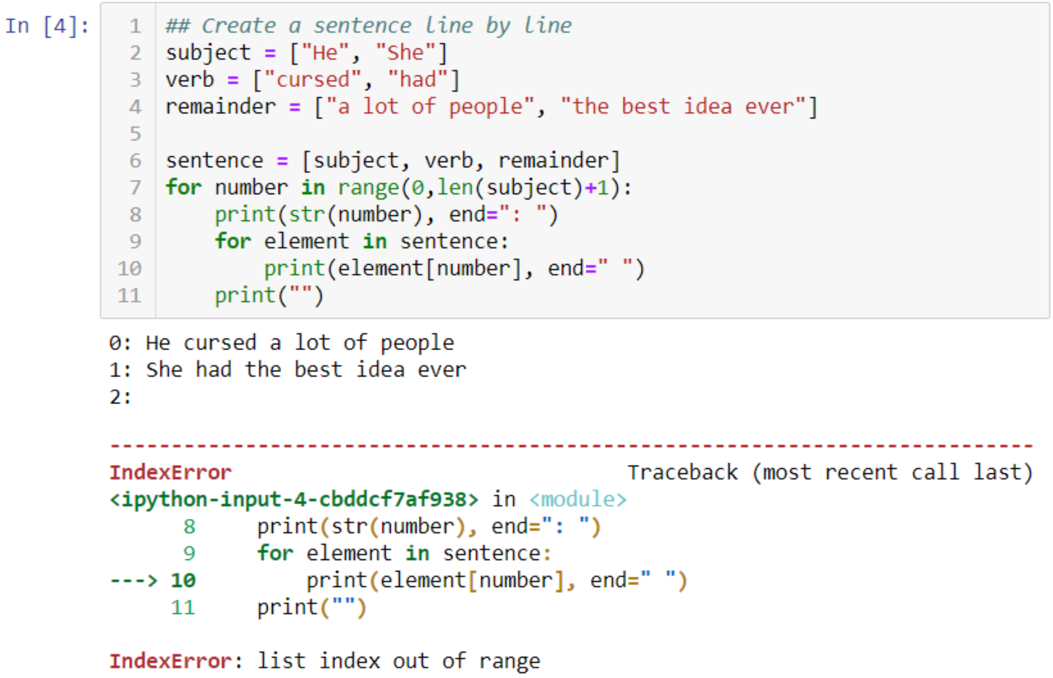
Figure B.3: Task 3, a missing colon



```
In [2]:
## Create a random sentence based on random picks
import random
subject = ["I", "He", "She", "It"]
verb = ["was", "did", "cursed", "imagined]
remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]

sentence = [subject, verb, remainder]
for element in sentence:
    print(element[random.randint(0,3)], end=" ")
print("")
```

```
  File "<ipython-input-2-78ee35f4efde>", line 4
    verb = ["was", "did", "cursed", "imagined]
                                              ^
SyntaxError: EOL while scanning string literal
```

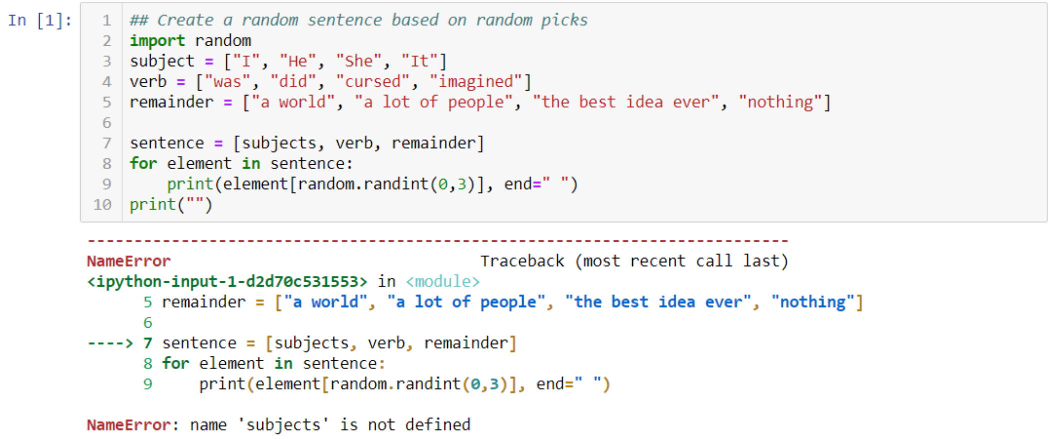
Figure B.4: Task 4, a non-terminated string



```
In [4]:
## Create a sentence line by line
subject = ["He", "She"]
verb = ["cursed", "had"]
remainder = ["a lot of people", "the best idea ever"]

sentence = [subject, verb, remainder]
for number in range(0,len(subject)+1):
    print(str(number), end=": ")
    for element in sentence:
        print(element[number], end=" ")
    print("")
```

```
0: He cursed a lot of people
1: She had the best idea ever
2:

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-cbddcf7af938> in <module>
      8     print(str(number), end=": ")
      9     for element in sentence:
---> 10         print(element[number], end=" ")
     11     print("")

IndexError: list index out of range
```

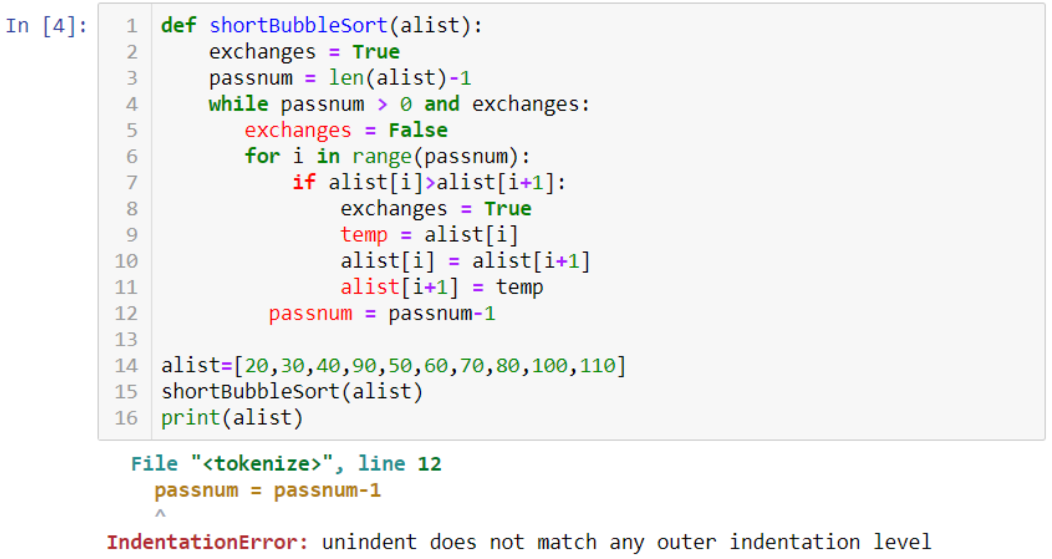
Figure B.5: Task 5, requesting an array position that's out of range



```
In [1]:
## Create a random sentence based on random picks
import random
subject = ["I", "He", "She", "It"]
verb = ["was", "did", "cursed", "imagined"]
remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]

sentence = [subjects, verb, remainder]
for element in sentence:
    print(element[random.randint(0,3)], end=" ")
print("")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-d2d70c531553> in <module>
      5 remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]
      6 
----> 7 sentence = [subjects, verb, remainder]
      8 for element in sentence:
      9     print(element[random.randint(0,3)], end=" ")

NameError: name 'subjects' is not defined
```

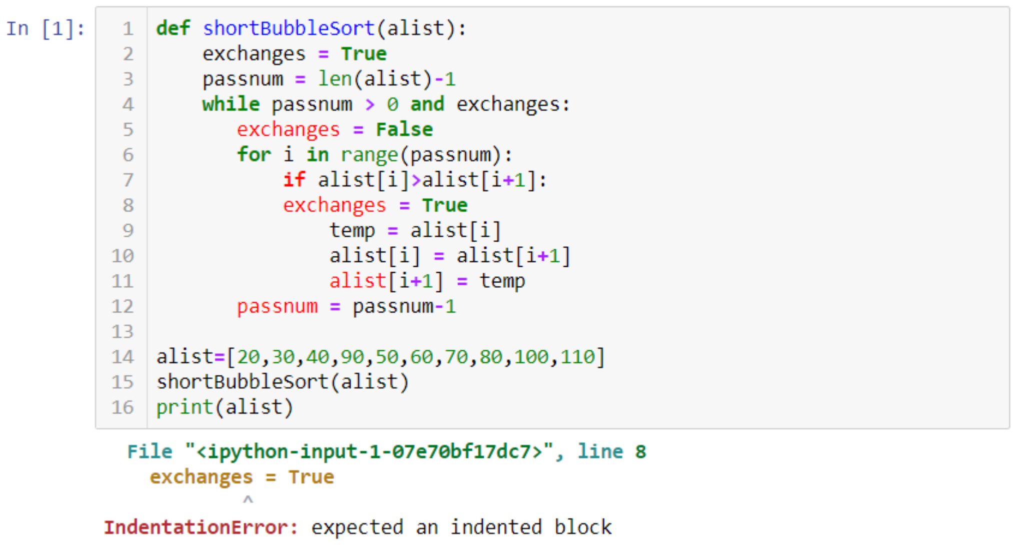
Figure B.6: Task 6, requesting data from a non-existing variable



```
In [4]:
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
         passnum = passnum-1

alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)
```

```
  File "<tokenize>", line 12
    passnum = passnum-1
    ^
IndentationError: unindent does not match any outer indentation level
```

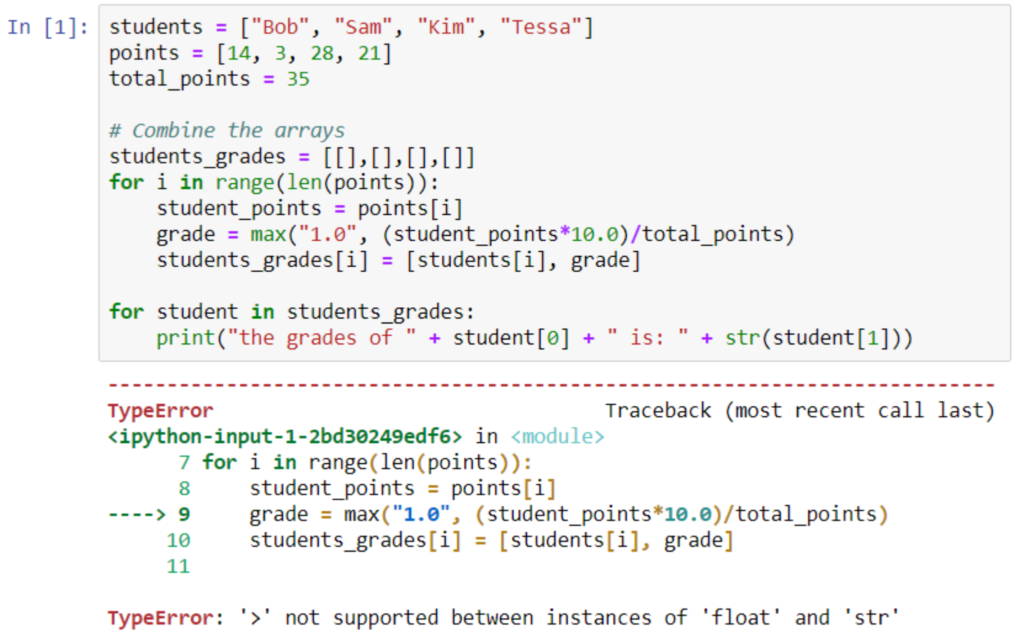
Figure B.7: Task 7, not a correct amount of indentation



```
In [1]:  1  def shortBubbleSort(alist):
         2      exchanges = True
         3      passnum = len(alist)-1
         4      while passnum > 0 and exchanges:
         5          exchanges = False
         6          for i in range(passnum):
         7              if alist[i]>alist[i+1]:
         8              exchanges = True
         9                  temp = alist[i]
        10                  alist[i] = alist[i+1]
        11                  alist[i+1] = temp
        12          passnum = passnum-1
        13
        14  alist=[20,30,40,90,50,60,70,80,100,110]
        15  shortBubbleSort(alist)
        16  print(alist)
```

```
  File "<ipython-input-1-07e70bf17dc7>", line 8
    exchanges = True
            ^
IndentationError: expected an indented block
```

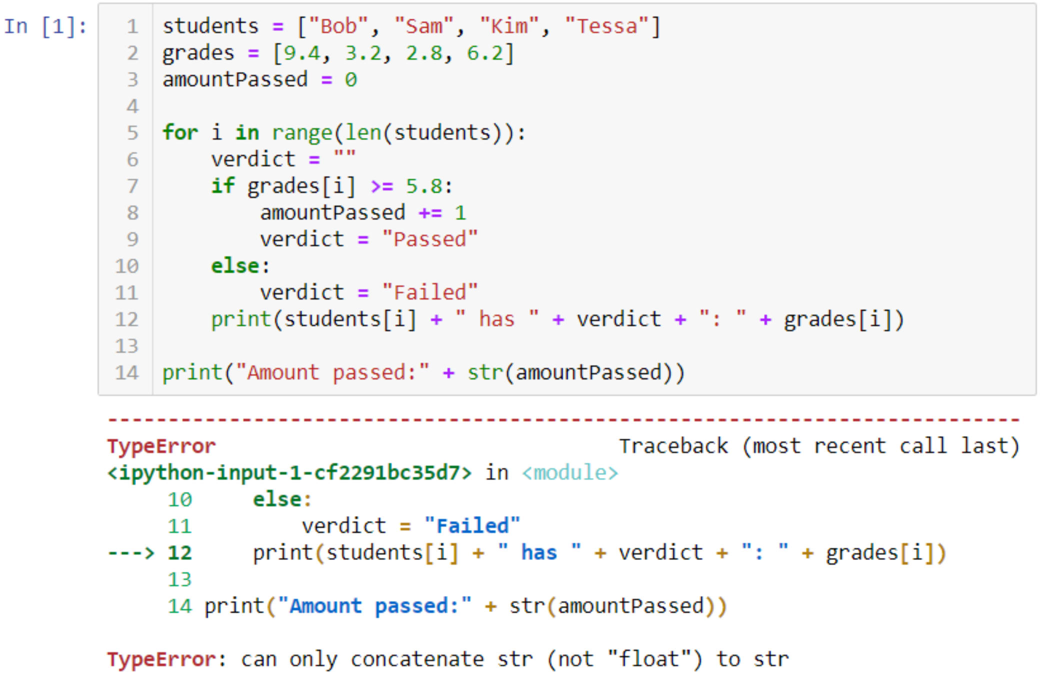
Figure B.8: Task 8, no indentation after an if-statement



```
In [1]: students = ["Bob", "Sam", "Kim", "Tessa"]
        points = [14, 3, 28, 21]
        total_points = 35

        # Combine the arrays
        students_grades = [[],[],[],[]]
        for i in range(len(points)):
            student_points = points[i]
            grade = max("1.0", (student_points*10.0)/total_points)
            students_grades[i] = [students[i], grade]

        for student in students_grades:
            print("the grades of " + student[0] + " is: " + str(student[1]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-2bd30249edf6> in <module>
      7 for i in range(len(points)):
      8     student_points = points[i]
----> 9     grade = max("1.0", (student_points*10.0)/total_points)
     10     students_grades[i] = [students[i], grade]
     11 

TypeError: '>' not supported between instances of 'float' and 'str'
```

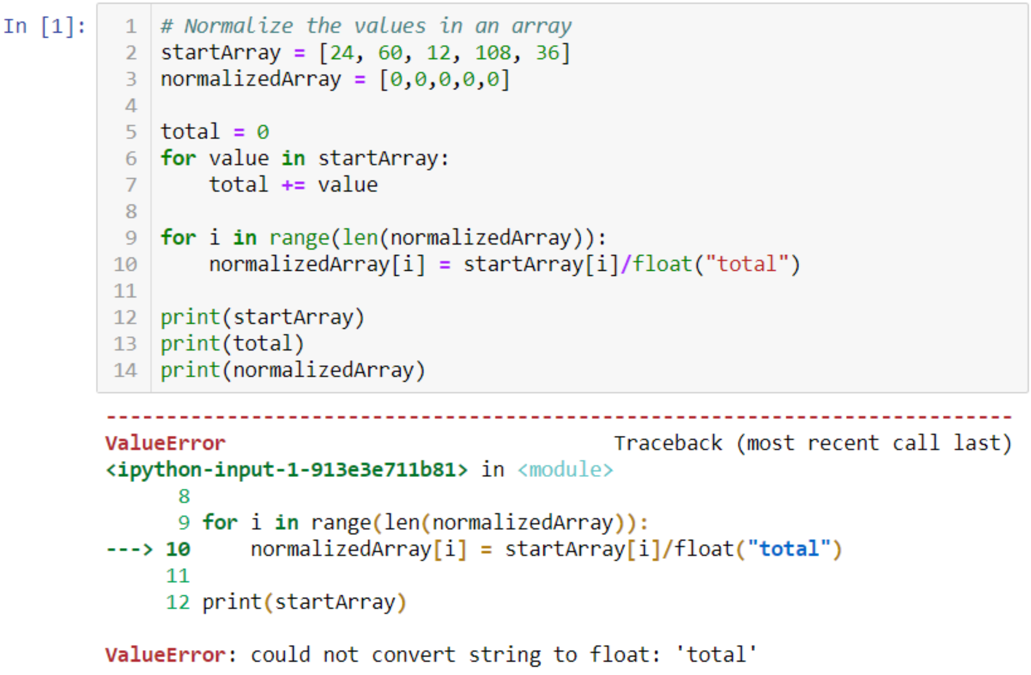
Figure B.9: Task 9, a parameter of the wrong type in a function call for function *max*



```
In [1]:
students = ["Bob", "Sam", "Kim", "Tessa"]
grades = [9.4, 3.2, 2.8, 6.2]
amountPassed = 0

for i in range(len(students)):
    verdict = ""
    if grades[i] >= 5.8:
        amountPassed += 1
        verdict = "Passed"
    else:
        verdict = "Failed"
    print(students[i] + " has " + verdict + ": " + grades[i])

print("Amount passed:" + str(amountPassed))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-cf2291bc35d7> in <module>
     10     else:
     11         verdict = "Failed"
---> 12     print(students[i] + " has " + verdict + ": " + grades[i])
     13 
     14 print("Amount passed:" + str(amountPassed))

TypeError: can only concatenate str (not "float") to str
```

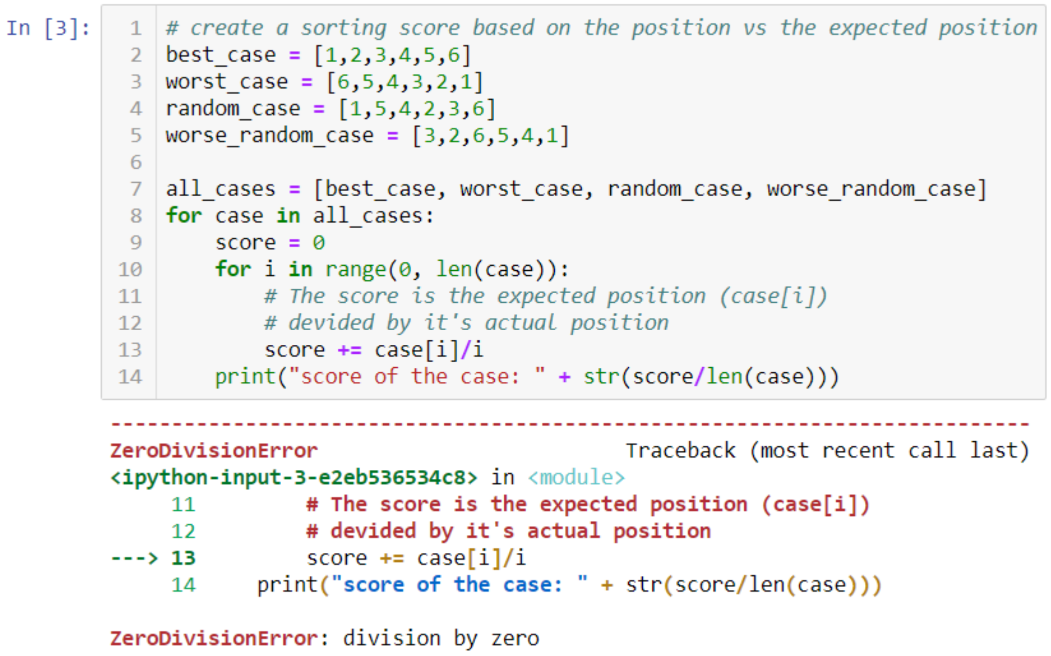
Figure B.10: Task 10, an operand of the wrong type in an operation (string concatenation)



```
In [1]:
# Normalize the values in an array
startArray = [24, 60, 12, 108, 36]
normalizedArray = [0,0,0,0,0]

total = 0
for value in startArray:
    total += value

for i in range(len(normalizedArray)):
    normalizedArray[i] = startArray[i]/float("total")

print(startArray)
print(total)
print(normalizedArray)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-913e3e711b81> in <module>
      8 
      9 for i in range(len(normalizedArray)):
---> 10     normalizedArray[i] = startArray[i]/float("total")
     11 
     12 print(startArray)

ValueError: could not convert string to float: 'total'
```

Figure B.11: Task 11, a parameter of the right type but an unexpected value in a function call



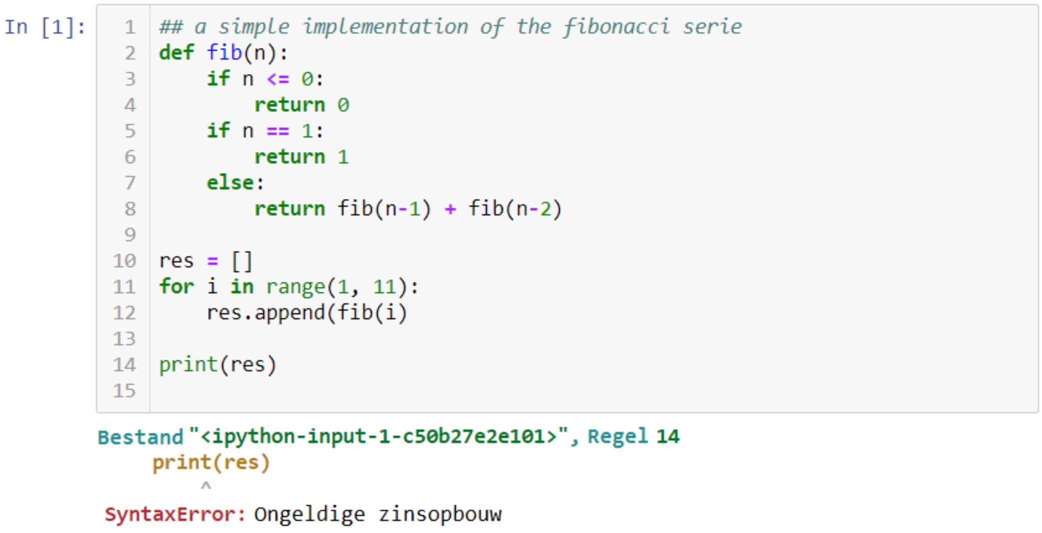
```
In [3]:
# create a sorting score based on the position vs the expected position
best_case = [1,2,3,4,5,6]
worst_case = [6,5,4,3,2,1]
random_case = [1,5,4,2,3,6]
worse_random_case = [3,2,6,5,4,1]

all_cases = [best_case, worst_case, random_case, worse_random_case]
for case in all_cases:
    score = 0
    for i in range(0, len(case)):
        # The score is the expected position (case[i])
        # devided by it's actual position
        score += case[i]/i
    print("score of the case: " + str(score/len(case)))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-e2eb536534c8> in <module>
     11         # The score is the expected position (case[i])
     12         # devided by it's actual position
---> 13         score += case[i]/i
     14     print("score of the case: " + str(score/len(case)))

ZeroDivisionError: division by zero
```

Figure B.12: Task 12, a division by zero

Appendix C

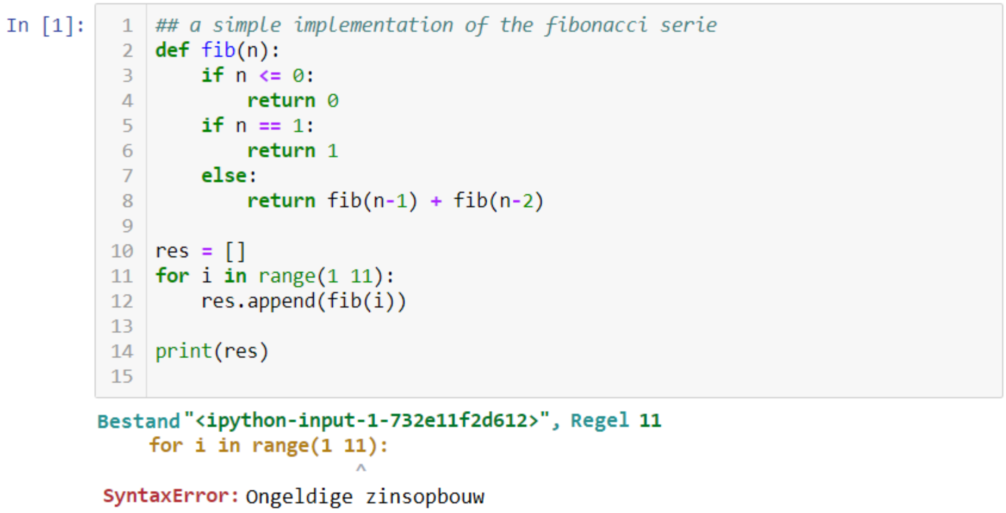
Tasks Translated



```
In [1]:
## a simple implementation of the fibonacci serie
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

res = []
for i in range(1, 11):
    res.append(fib(i))

print(res)

```

```
Bestand "<ipython-input-1-c50b27e2e101>", Regel 14
    print(res)
         ^
 SyntaxError: Ongeldige zinsopbouw
```

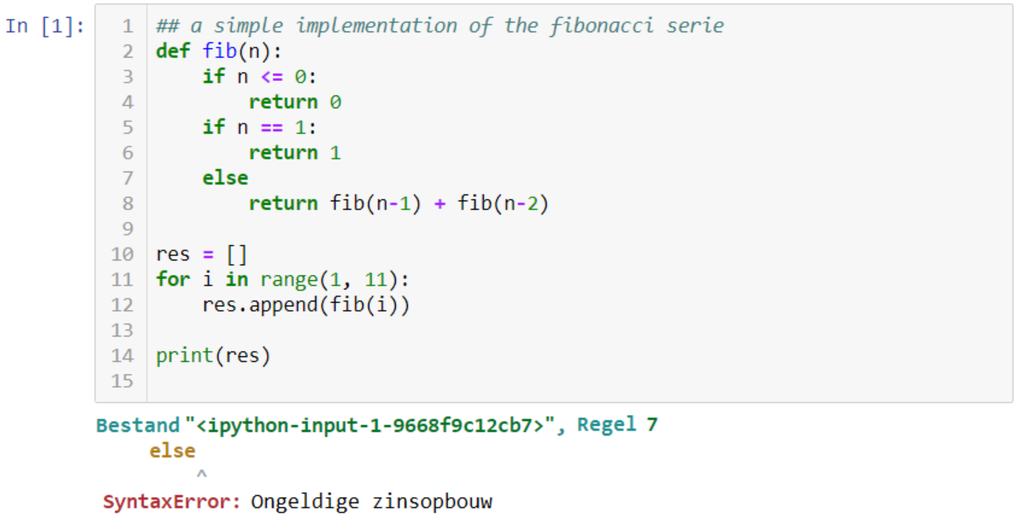
Figure C.1: Task 1 with error message translated to Dutch



```
In [1]:
## a simple implementation of the fibonacci serie
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

res = []
for i in range(1 11):
    res.append(fib(i))

print(res)

```

```
Bestand "<ipython-input-1-732e11f2d612>", Regel 11
    for i in range(1 11):
                      ^
SyntaxError: Ongeldige zinsopbouw
```

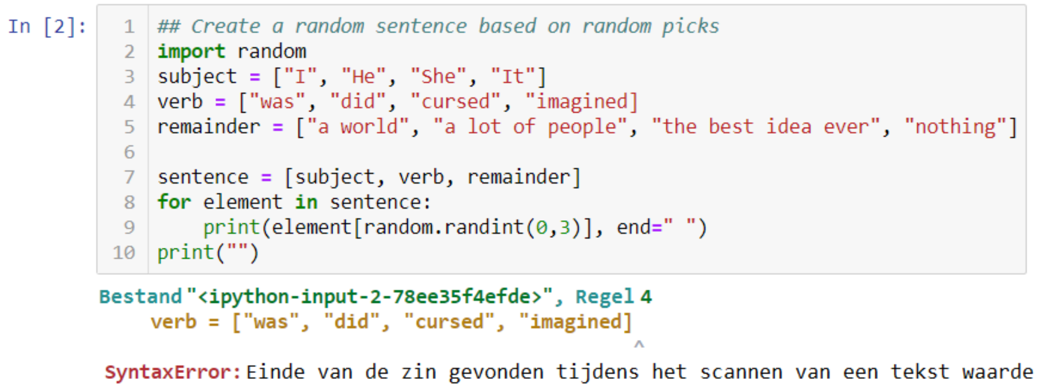
Figure C.2: Task 2 with error message translated to Dutch



```
In [1]:
## a simple implementation of the fibonacci serie
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else
        return fib(n-1) + fib(n-2)

res = []
for i in range(1, 11):
    res.append(fib(i))

print(res)

```

```
Bestand "<ipython-input-1-9668f9c12cb7>", Regel 7
    else
        ^
SyntaxError: Ongeldige zinsopbouw
```

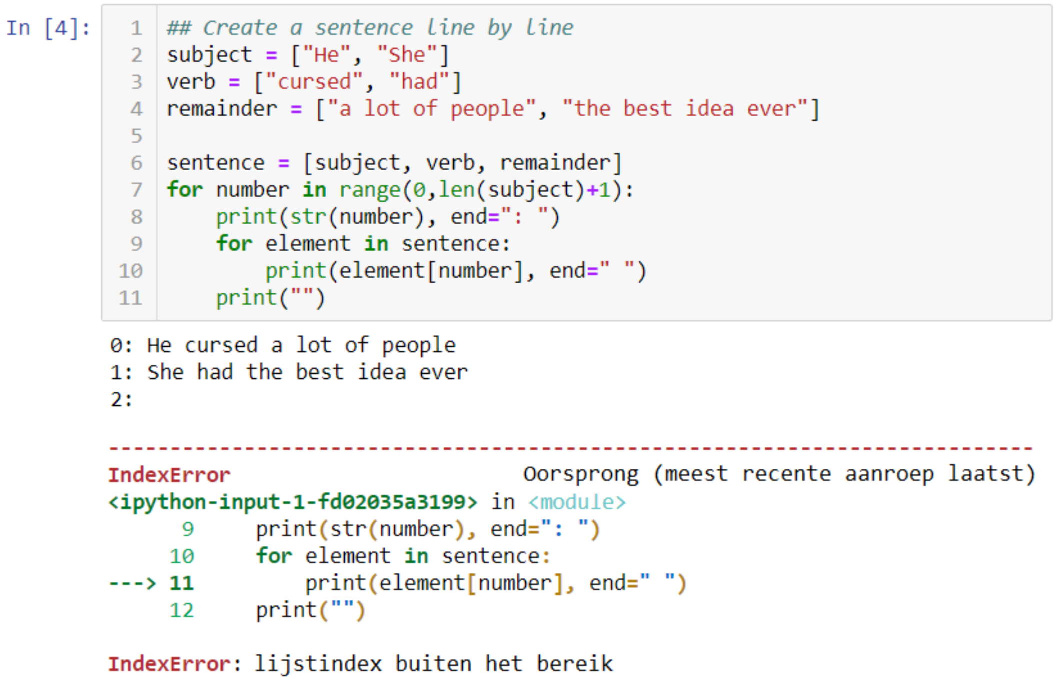
Figure C.3: Task 3 with error message translated to Dutch



```
In [2]:
## Create a random sentence based on random picks
import random
subject = ["I", "He", "She", "It"]
verb = ["was", "did", "cursed", "imagined]
remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]

sentence = [subject, verb, remainder]
for element in sentence:
    print(element[random.randint(0,3)], end=" ")
print("")
```

```
Bestand "<ipython-input-2-78ee35f4efde>", Regel 4
    verb = ["was", "did", "cursed", "imagined]
                                              ^
SyntaxError:Einde van de zin gevonden tijdens het scannen van een tekst waarde
```

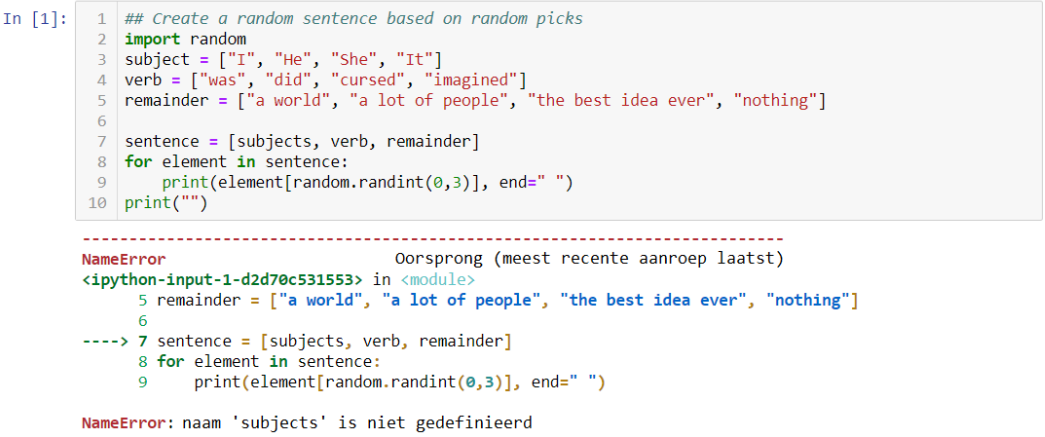
Figure C.4: Task 4 with error message translated to Dutch



```
In [4]:
## Create a sentence line by line
subject = ["He", "She"]
verb = ["cursed", "had"]
remainder = ["a lot of people", "the best idea ever"]

sentence = [subject, verb, remainder]
for number in range(0,len(subject)+1):
    print(str(number), end=": ")
    for element in sentence:
        print(element[number], end=" ")
    print("")
```

```
0: He cursed a lot of people
1: She had the best idea ever
2: 

---------------------------------------------------------------------------
IndexError                                Oorsprong (meest recente aanroep laatst)
<ipython-input-1-fd02035a3199> in <module>
      9     print(str(number), end=": ")
     10     for element in sentence:
---> 11         print(element[number], end=" ")
     12     print("")

IndexError: lijstindex buiten het bereik
```

Figure C.5: Task 5 with error message translated to Dutch

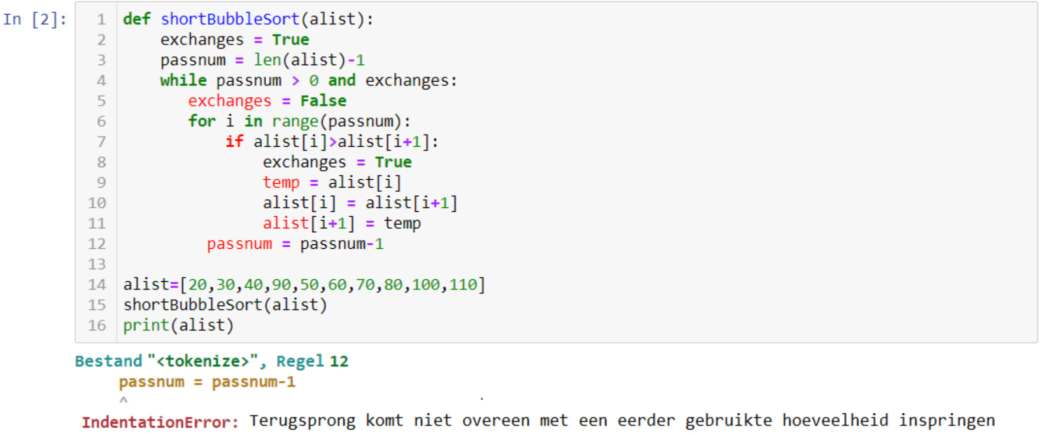


In [1]:

```
## Create a random sentence based on random picks
import random
subject = ["I", "He", "She", "It"]
verb = ["was", "did", "cursed", "imagined"]
remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]

sentence = [subjects, verb, remainder]
for element in sentence:
    print(element[random.randint(0,3)], end=" ")
print("")
```

```
---------------------------------------------------------------------------
NameError                                 Oorsprong (meest recente aanroep laatst)
<ipython-input-1-d2d70c531553> in <module>
      5 remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]
      6 
----> 7 sentence = [subjects, verb, remainder]
      8 for element in sentence:
      9     print(element[random.randint(0,3)], end=" ")

NameError: naam 'subjects' is niet gedefinieerd
```

Figure C.6: Task 6 with error message translated to Dutch

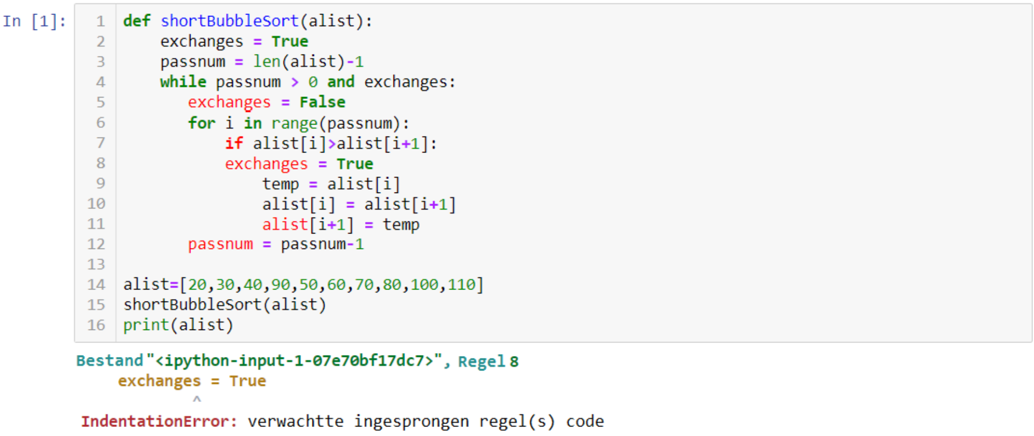


In [2]:

```
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
         passnum = passnum-1

alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)
```

```
Bestand "<tokenize>", Regel 12
    passnum = passnum-1
    ^
IndentationError: Terugsprong komt niet overeen met een eerder gebruikte hoeveelheid inspringen
```

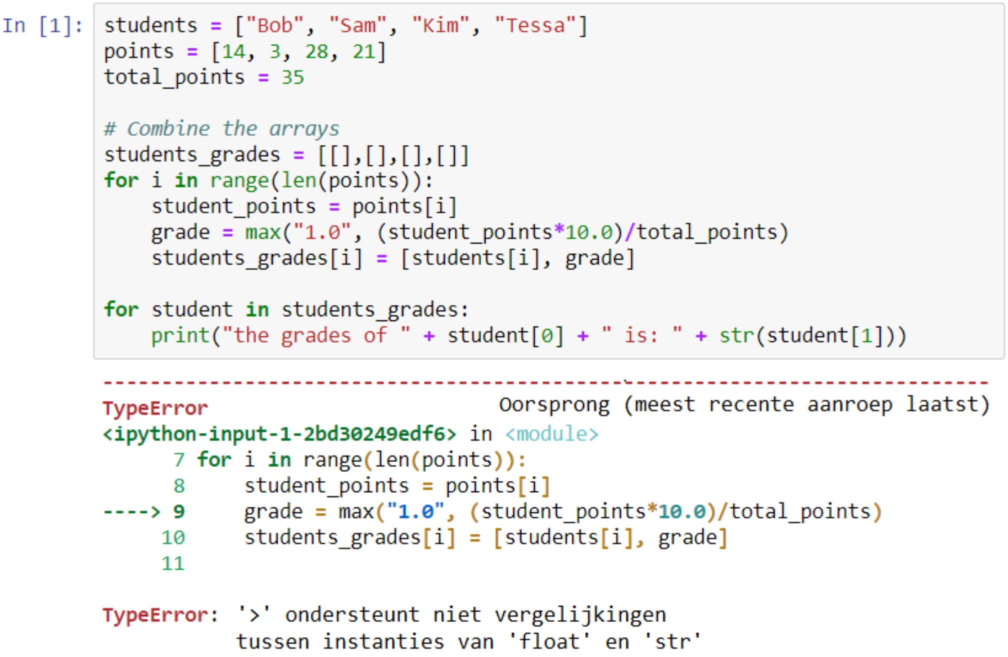
Figure C.7: Task 7 with error message translated to Dutch



```
In [1]:  def shortBubbleSort(alist):
             exchanges = True
             passnum = len(alist)-1
             while passnum > 0 and exchanges:
                exchanges = False
                for i in range(passnum):
                    if alist[i]>alist[i+1]:
                    exchanges = True
                        temp = alist[i]
                        alist[i] = alist[i+1]
                        alist[i+1] = temp
                passnum = passnum-1

         alist=[20,30,40,90,50,60,70,80,100,110]
         shortBubbleSort(alist)
         print(alist)
```

```
Bestand "<ipython-input-1-07e70bf17dc7>", Regel 8
    exchanges = True
            ^
IndentationError: verwachtte ingesprongen regel(s) code
```

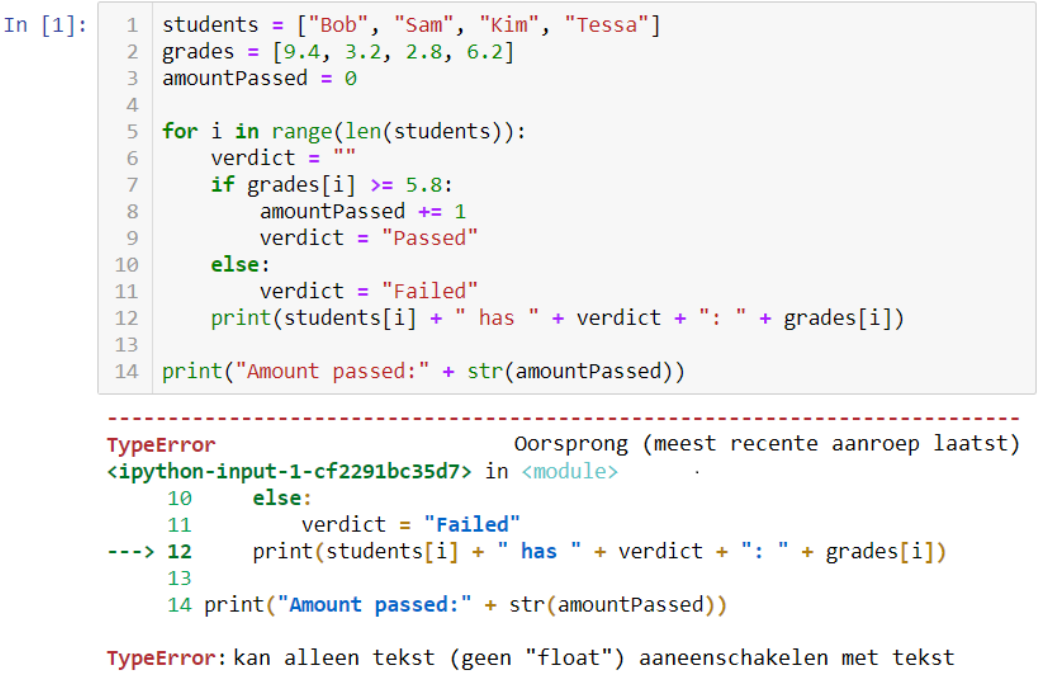
Figure C.8: Task 8 with error message translated to Dutch



```
In [1]:  students = ["Bob", "Sam", "Kim", "Tessa"]
         points = [14, 3, 28, 21]
         total_points = 35

         # Combine the arrays
         students_grades = [[],[],[],[]]
         for i in range(len(points)):
             student_points = points[i]
             grade = max("1.0", (student_points*10.0)/total_points)
             students_grades[i] = [students[i], grade]

         for student in students_grades:
             print("the grades of " + student[0] + " is: " + str(student[1]))
```

```
---------------------------------------------------------------------------
TypeError                                 Oorsprong (meest recente aanroep laatst)
<ipython-input-1-2bd30249edf6> in <module>
      7 for i in range(len(points)):
      8     student_points = points[i]
----> 9     grade = max("1.0", (student_points*10.0)/total_points)
     10     students_grades[i] = [students[i], grade]
     11

TypeError: '>' ondersteunt niet vergelijkingen
           tussen instanties van 'float' en 'str'
```

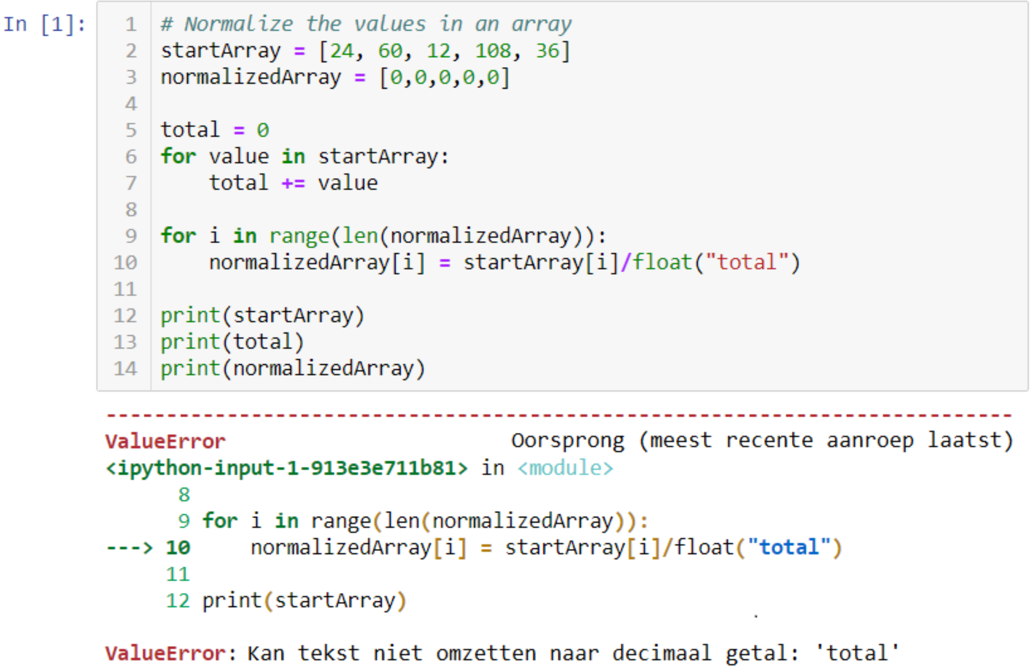
Figure C.9: Task 9 with error message translated to Dutch



```
In [1]:
students = ["Bob", "Sam", "Kim", "Tessa"]
grades = [9.4, 3.2, 2.8, 6.2]
amountPassed = 0

for i in range(len(students)):
    verdict = ""
    if grades[i] >= 5.8:
        amountPassed += 1
        verdict = "Passed"
    else:
        verdict = "Failed"
    print(students[i] + " has " + verdict + ": " + grades[i])

print("Amount passed:" + str(amountPassed))
```

```
---------------------------------------------------------------------------
TypeError                                 Oorsprong (meest recente aanroep laatst)
<ipython-input-1-cf2291bc35d7> in <module>
     10     else:
     11         verdict = "Failed"
---> 12     print(students[i] + " has " + verdict + ": " + grades[i])
     13 
     14 print("Amount passed:" + str(amountPassed))

TypeError: kan alleen tekst (geen "float") aaneenschakelen met tekst
```

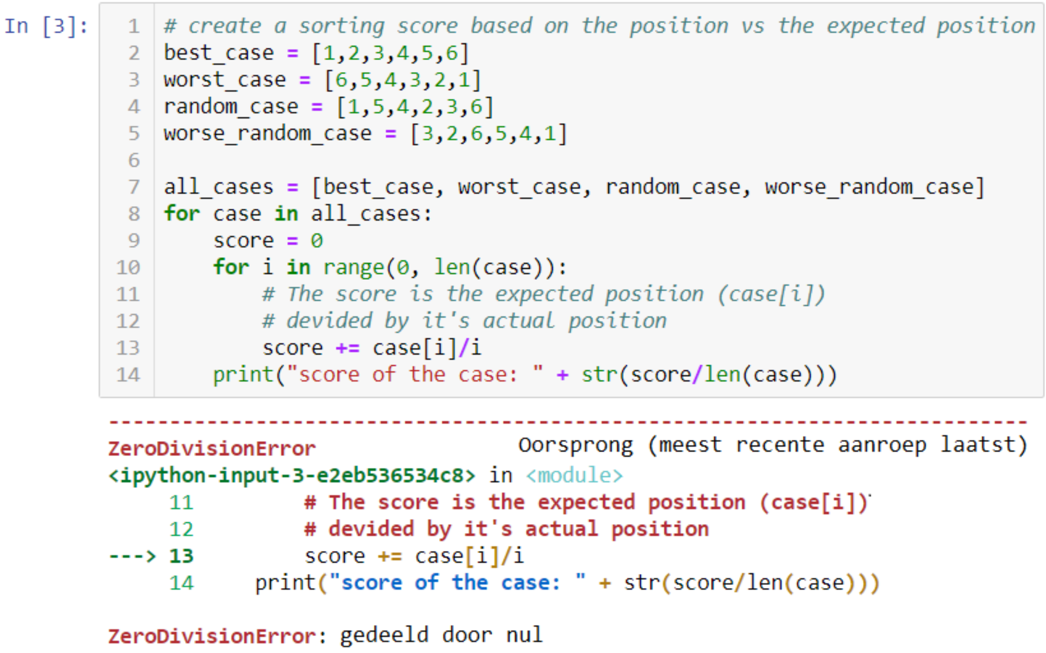
Figure C.10: Task 10 with error message translated to Dutch



```
In [1]:
# Normalize the values in an array
startArray = [24, 60, 12, 108, 36]
normalizedArray = [0,0,0,0,0]

total = 0
for value in startArray:
    total += value

for i in range(len(normalizedArray)):
    normalizedArray[i] = startArray[i]/float("total")

print(startArray)
print(total)
print(normalizedArray)
```

```
---------------------------------------------------------------------------
ValueError                                Oorsprong (meest recente aanroep laatst)
<ipython-input-1-913e3e711b81> in <module>
      8 
      9 for i in range(len(normalizedArray)):
---> 10     normalizedArray[i] = startArray[i]/float("total")
     11 
     12 print(startArray)

ValueError: Kan tekst niet omzetten naar decimaal getal: 'total'
```

Figure C.11: Task 11 with error message translated to Dutch



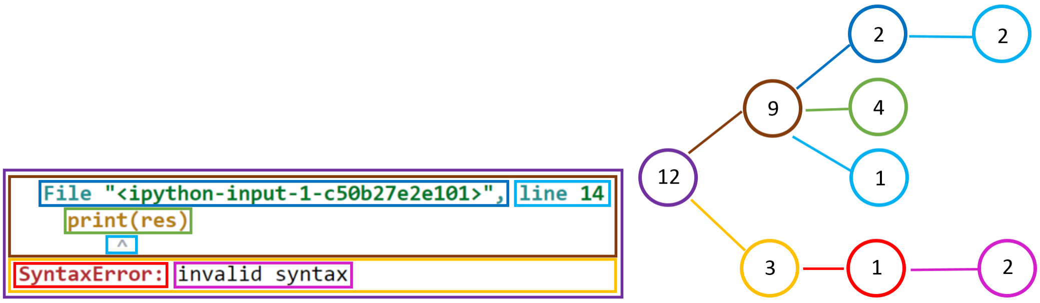
```
In [3]:
# create a sorting score based on the position vs the expected position
best_case = [1,2,3,4,5,6]
worst_case = [6,5,4,3,2,1]
random_case = [1,5,4,2,3,6]
worse_random_case = [3,2,6,5,4,1]

all_cases = [best_case, worst_case, random_case, worse_random_case]
for case in all_cases:
    score = 0
    for i in range(0, len(case)):
        # The score is the expected position (case[i])
        # devided by it's actual position
        score += case[i]/i
    print("score of the case: " + str(score/len(case)))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Oorsprong (meest recente aanroep laatst)
<ipython-input-3-e2eb536534c8> in <module>
     11         # The score is the expected position (case[i])
     12         # devided by it's actual position
---> 13         score += case[i]/i
     14     print("score of the case: " + str(score/len(case)))

ZeroDivisionError: gedeeld door nul
```

Figure C.12: Task 12 with error message translated to Dutch

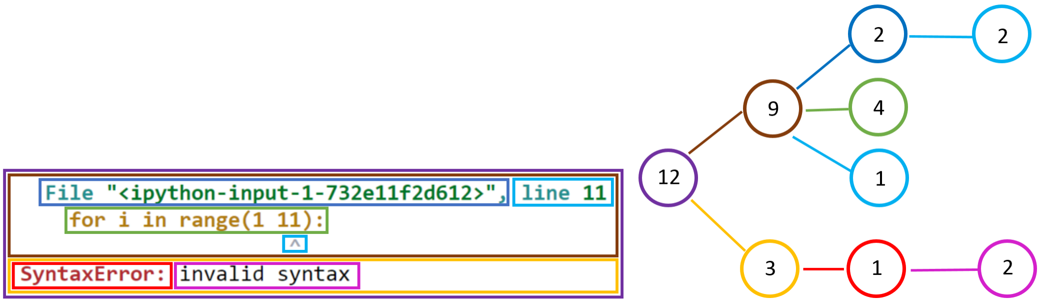
Appendix D

EMCF Applied



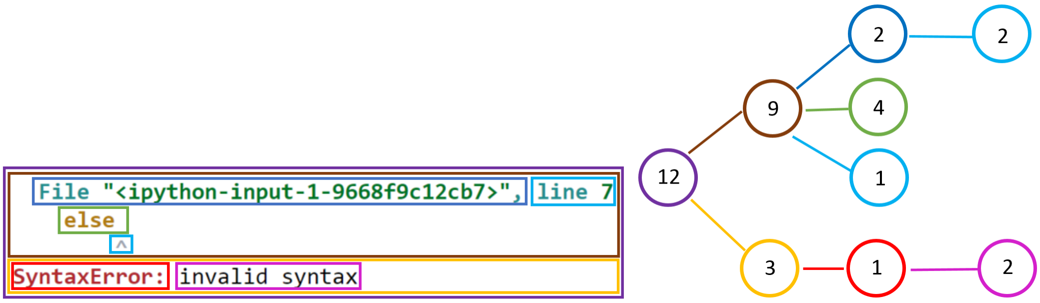
```
  File "<ipython-input-1-c50b27e2e101>", line 14
    print(res)
       ^
SyntaxError: invalid syntax
```

Figure D.1: EMCF applied to Task 1



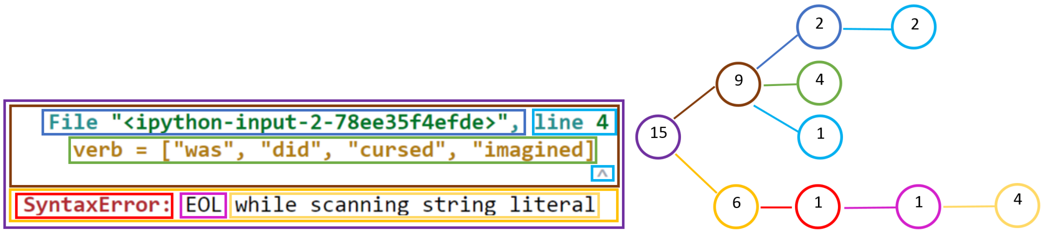
```
  File "<ipython-input-1-732e11f2d612>", line 11
    for i in range(1 11):
                     ^
SyntaxError: invalid syntax
```

Figure D.2: EMCF applied to Task 2



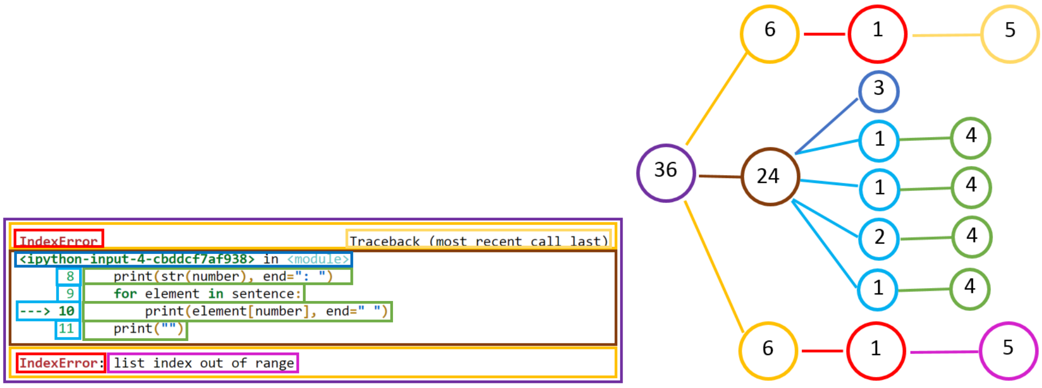
```
  File "<ipython-input-1-9668f9c12cb7>", line 7
    else
       ^
SyntaxError: invalid syntax
```

Figure D.3: EMCF applied to Task 3



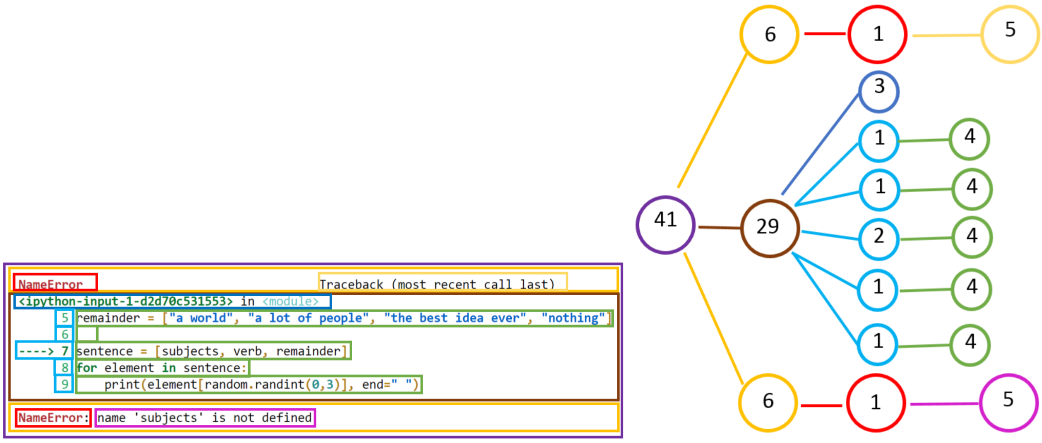
```
  File "<ipython-input-2-78ee35f4efde>", line 4
    verb = ["was", "did", "cursed", "imagined]
                                              ^
SyntaxError: EOL while scanning string literal
```

Figure D.4: EMCF applied to Task 4



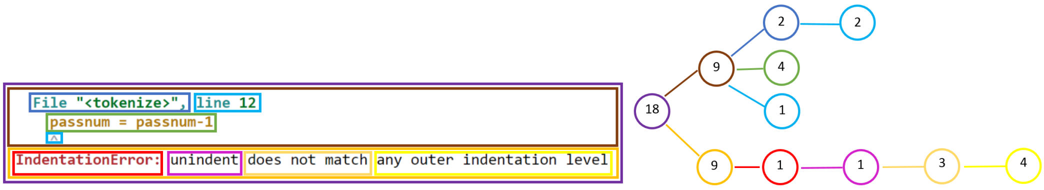
```
IndexError                                Traceback (most recent call last)
<ipython-input-4-cbddcf7af938> in <module>
      8     print(str(number), end=": ")
      9     for element in sentence:
---> 10         print(element[number], end=" ")
     11     print("")

IndexError: list index out of range
```

Figure D.5: EMCF applied to Task 5



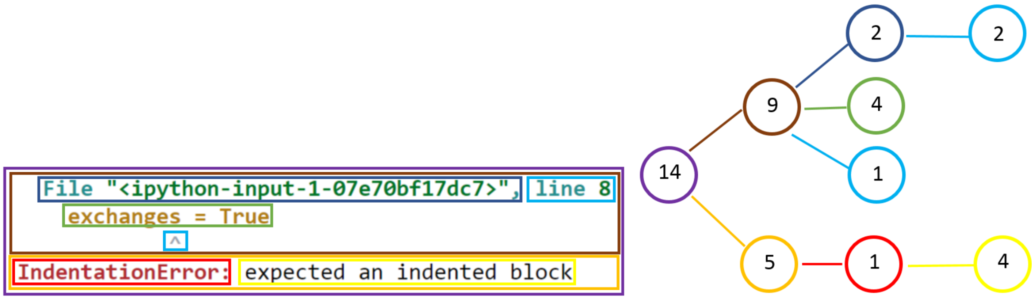
```
NameError                                 Traceback (most recent call last)
<ipython-input-1-d2d70c531553> in <module>
      5 remainder = ["a world", "a lot of people", "the best idea ever", "nothing"]
      6 
----> 7 sentence = [subjects, verb, remainder]
      8 for element in sentence:
      9     print(element[random.randint(0,3)], end=" ")

NameError: name 'subjects' is not defined
```

Figure D.6: EMCF applied to Task 6



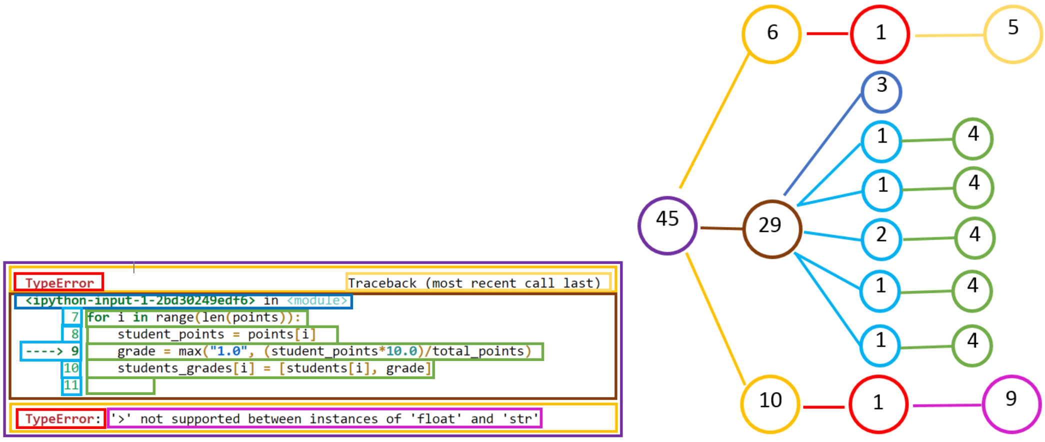
```
  File "<tokenize>", line 12
    passnum = passnum-1
    ^
IndentationError: unindent does not match any outer indentation level
```

Figure D.7: EMCF applied to Task 7



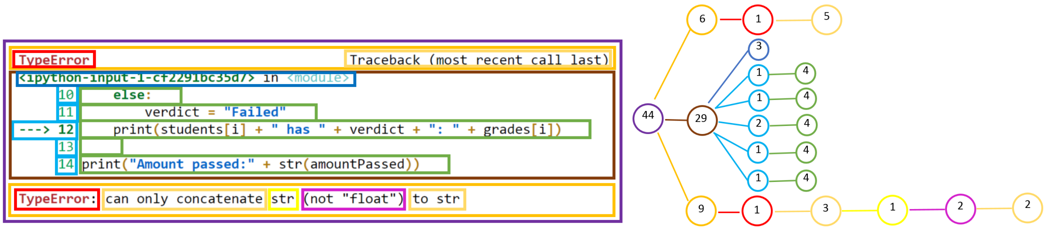
```
  File "<ipython-input-1-07e70bf17dc7>", line 8
    exchanges = True
              ^
IndentationError: expected an indented block
```

Figure D.8: EMCF applied to Task 8



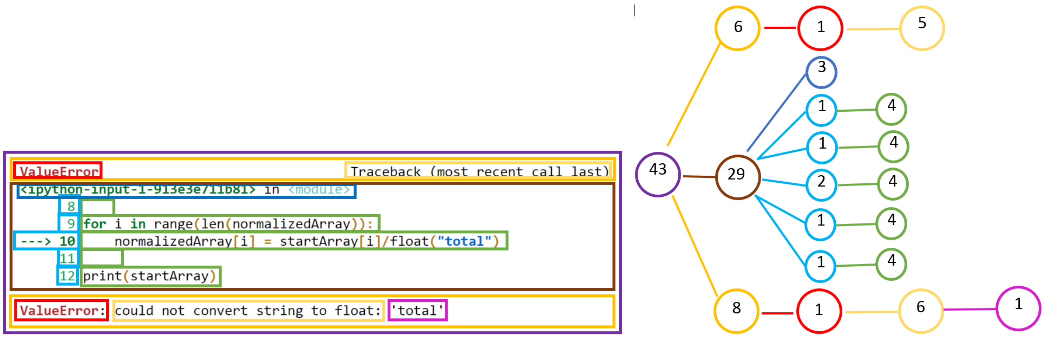
```
TypeError                                 Traceback (most recent call last)
<ipython-input-1-2bd30249edf6> in <module>
      7 for i in range(len(points)):
      8     student_points = points[i]
----> 9     grade = max("1.0", (student_points*10.0)/total_points)
     10     students_grades[i] = [students[i], grade]
     11

TypeError: '>' not supported between instances of 'float' and 'str'
```

Figure D.9: EMCF applied to Task 9



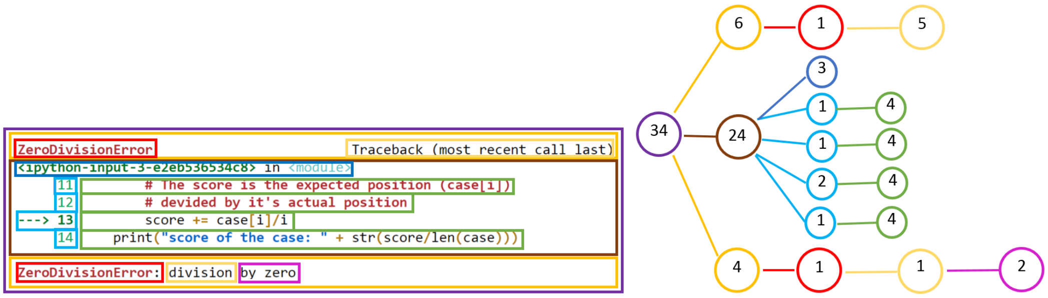
```
TypeError                                 Traceback (most recent call last)
<ipython-input-1-cf2291bc35d7> in <module>
     10     else:
     11         verdict = "Failed"
---> 12     print(students[i] + " has " + verdict + ": " + grades[i])
     13
     14 print("Amount passed:" + str(amountPassed))

TypeError: can only concatenate str (not "float") to str
```

Figure D.10: EMCF applied to Task 10



```
ValueError                                Traceback (most recent call last)
<ipython-input-1-913e3e711b81> in <module>
      8
      9 for i in range(len(normalizedArray)):
---> 10     normalizedArray[i] = startArray[i]/float("total")
     11
     12 print(startArray)

ValueError: could not convert string to float: 'total'
```

Figure D.11: EMCF applied to Task 11



```
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-e2eb536534c8> in <module>
     11         # The score is the expected position (case[i])
     12         # devided by it's actual position
---> 13         score += case[i]/i
     14     print("score of the case: " + str(score/len(case)))

ZeroDivisionError: division by zero
```

Figure D.12: EMCF applied to Task 12

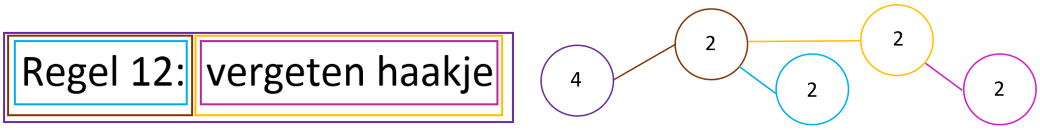
Appendix E

Error Messages of the Participants

This chapter shows all error messages made by the participants. Each message is analysed with the EMCF.

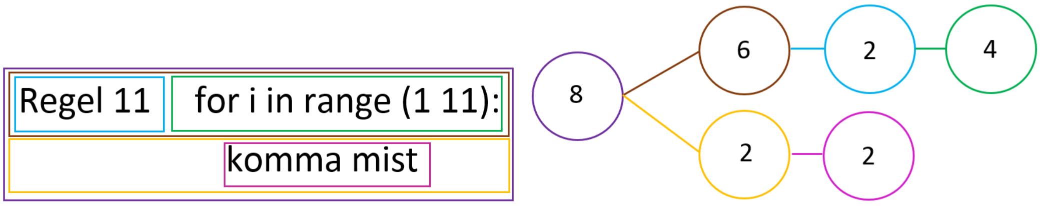
E.1 Participant 1

Participant 1 was not able to produce an error message for task 7.



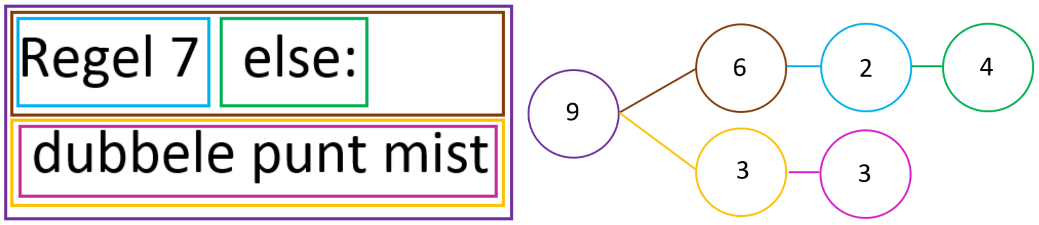
Regel 12: vergeten haakje

Figure E.1: EMCF applied to the error message created by participant 1 for task 1



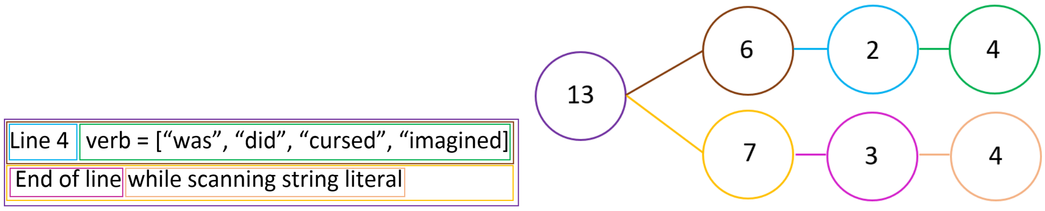
Regel 11 for i in range (1 11):
komma mist

Figure E.2: EMCF applied to the error message created by participant 1 for task 2



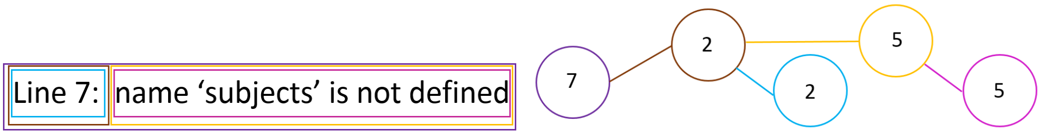
Regel 7 else:
dubbele punt mist

Figure E.3: EMCF applied to the error message created by participant 1 for task 3



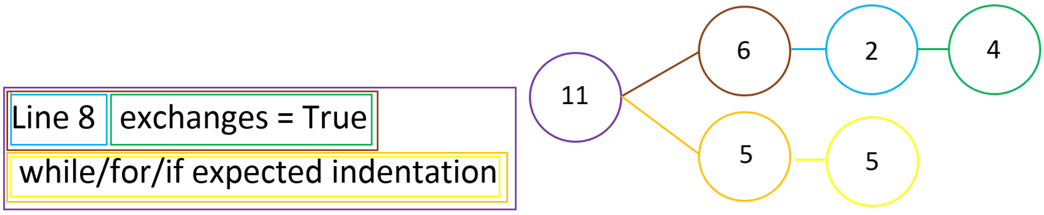
Line 4 verb = [“was”, “did”, “cursed”, “imagined]
End of line while scanning string literal

Figure E.4: EMCF applied to the error message created by participant 1 for task 4



Line 7: name ‘subjects’ is not defined

Figure E.5: EMCF applied to the error message created by participant 1 for task 6



Line 8 exchanges = True
while/for/if expected indentation

Figure E.6: EMCF applied to the error message created by participant 1 for task 8

E.2 Participant 2

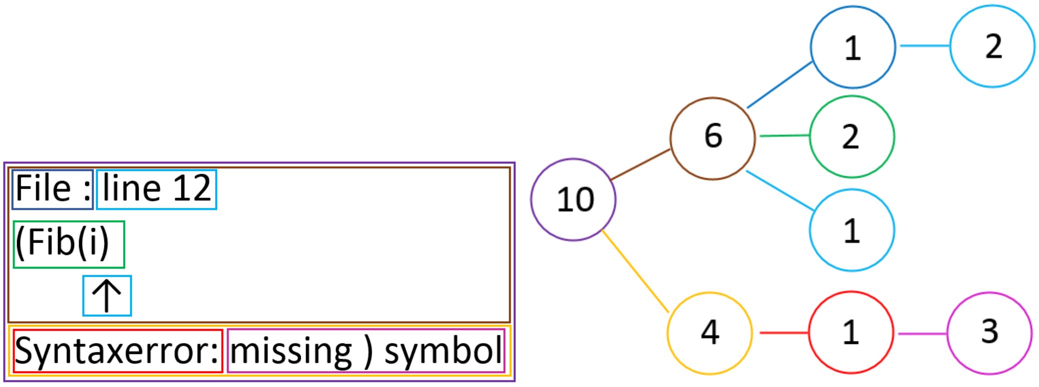


Figure E.7: EMCF applied to the error message created by participant 2 for task 1

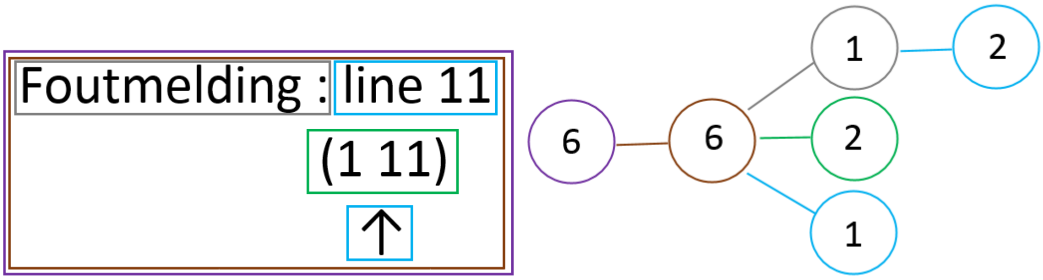


Figure E.8: EMCF applied to the error message created by participant 2 for task 2

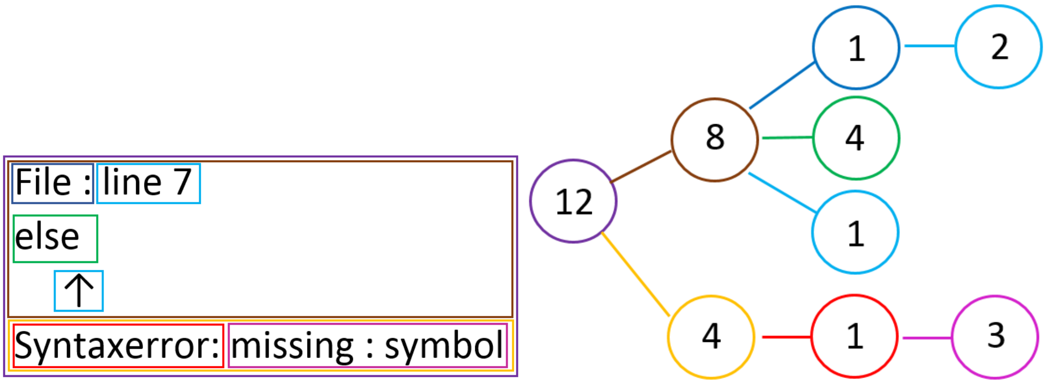
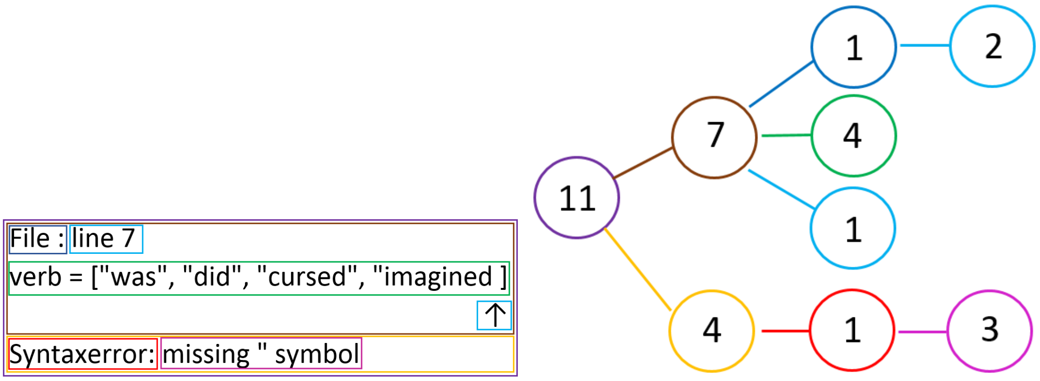
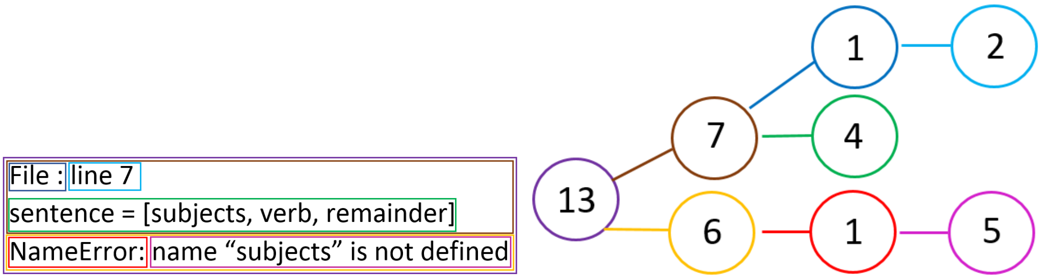


Figure E.9: EMCF applied to the error message created by participant 2 for task 3



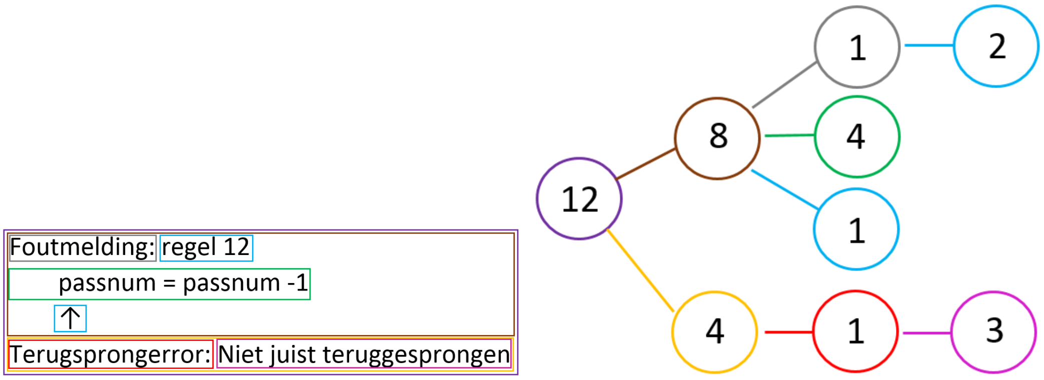
```
File : line 7
verb = ["was", "did", "cursed", "imagined ]
                                          ↑
Syntaxerror: missing " symbol
```

Figure E.10: EMCF applied to the error message created by participant 2 for task 4



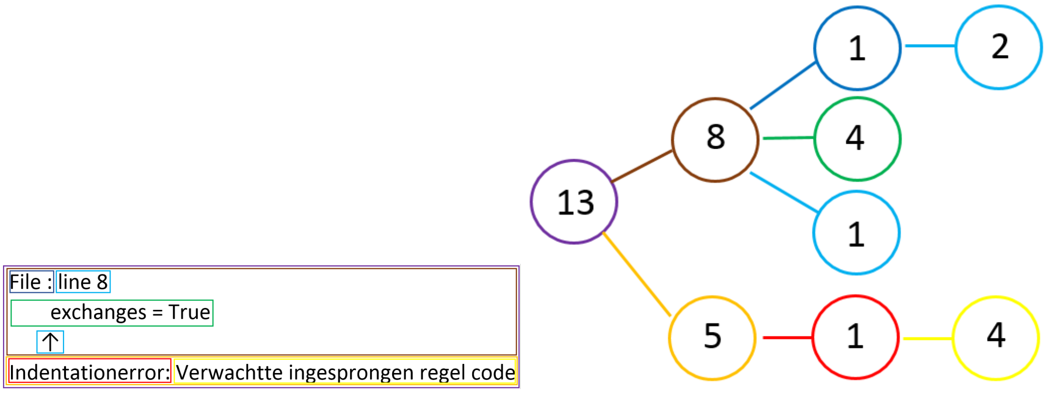
```
File : line 7
sentence = [subjects, verb, remainder]
NameError: name “subjects” is not defined
```

Figure E.11: EMCF applied to the error message created by participant 2 for task 6



```
Foutmelding: regel 12
      passnum = passnum -1
      ↑
Terugsprongerror: Niet juist teruggesprongen
```

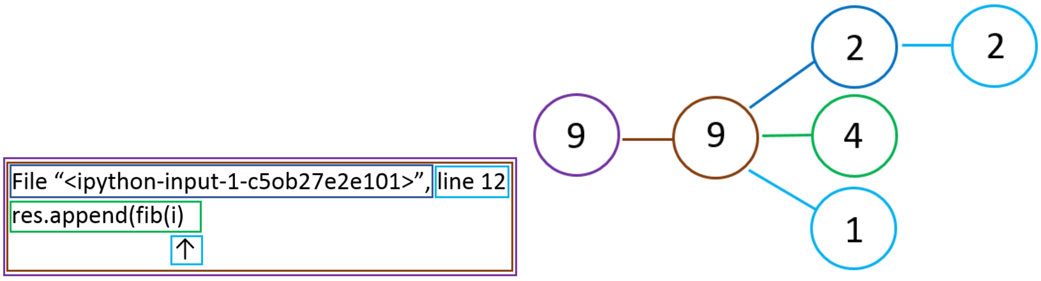
Figure E.12: EMCF applied to the error message created by participant 2 for task 7



```
File : line 8
      exchanges = True
   ↑
Indentationerror: Verwachtte ingesprongen regel code
```

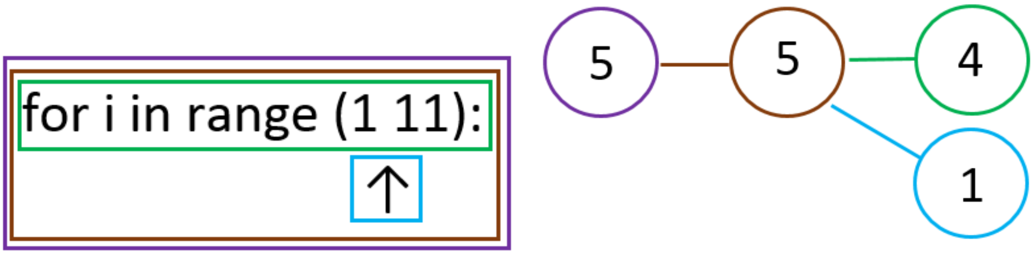
Figure E.13: EMCF applied to the error message created by participant 2 for task 8

E.3 Participant 3



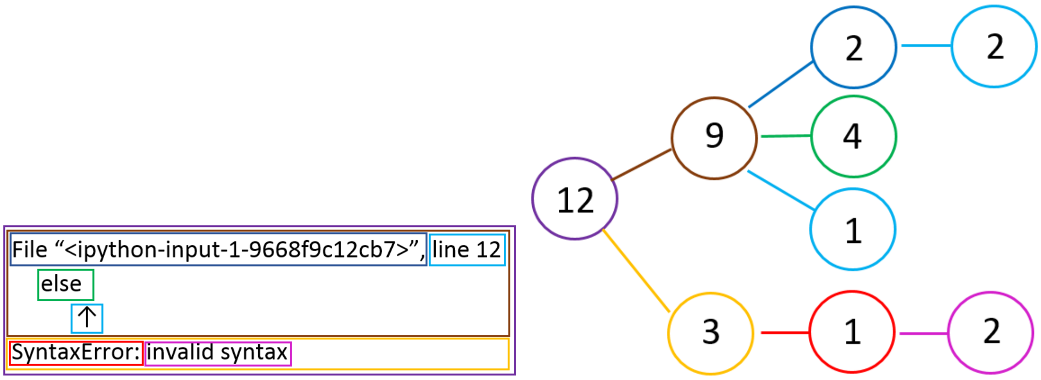
```
File “<ipython-input-1-c5ob27e2e101>”, line 12
res.append(fib(i)
                 ↑
```

Figure E.14: EMCF applied to the error message created by participant 3 for task 1



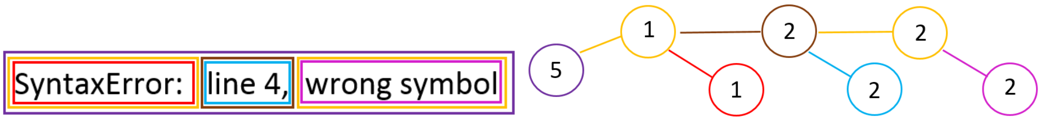
```
for i in range (1 11):
                  ↑
```

Figure E.15: EMCF applied to the error message created by participant 3 for task 2



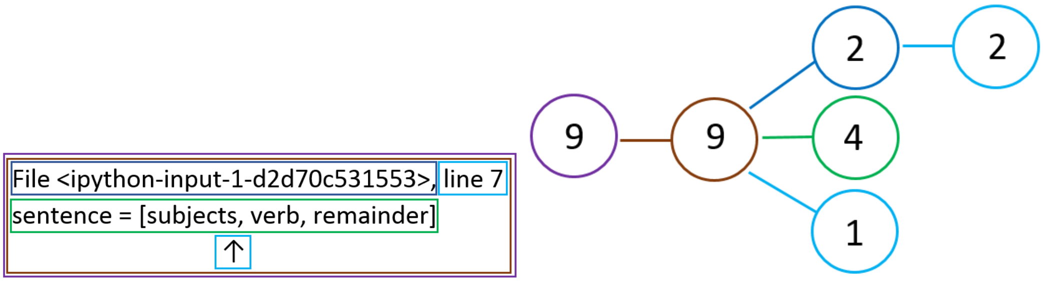
```
File “<ipython-input-1-9668f9c12cb7>”, line 12
    else
       ↑
SyntaxError: invalid syntax
```

Figure E.16: EMCF applied to the error message created by participant 3 for task 3



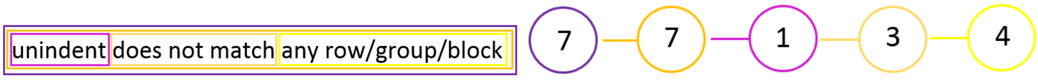
```
SyntaxError: line 4, wrong symbol
```

Figure E.17: EMCF applied to the error message created by participant 3 for task 4



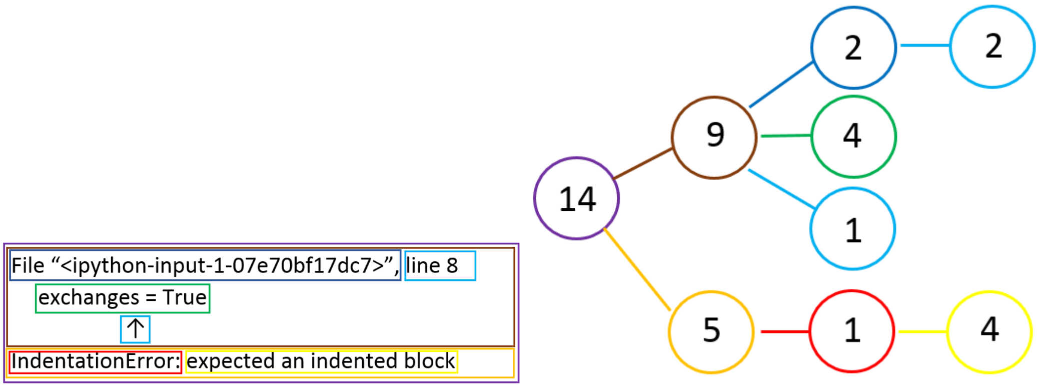
```
File <ipython-input-1-d2d70c531553>, line 7
sentence = [subjects, verb, remainder]
                       ↑
```

Figure E.18: EMCF applied to the error message created by participant 3 for task 6



```
unindent does not match any row/group/block
```

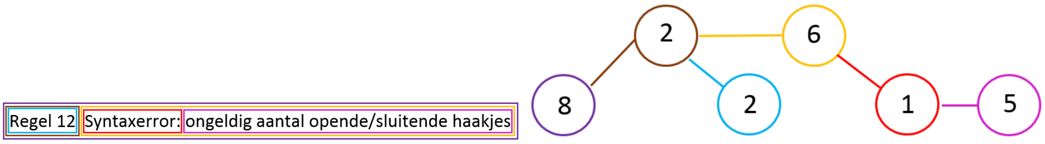
Figure E.19: EMCF applied to the error message created by participant 3 for task 7



```
File "<ipython-input-1-07e70bf17dc7>", line 8
    exchanges = True
            ↑
IndentationError: expected an indented block
```

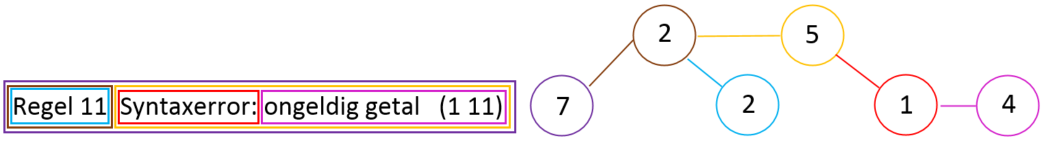
Figure E.20: EMCF applied to the error message created by participant 3 for task 8

E.4 Participant 4



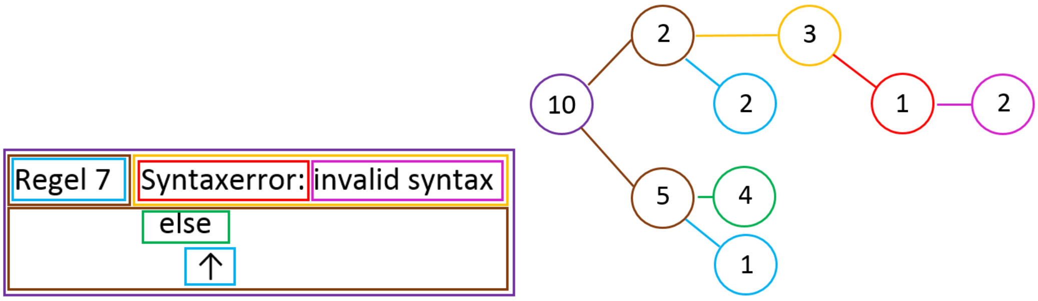
```
Regel 12 Syntaxerror: ongeldig aantal opende/sluitende haakjes
```

Figure E.21: EMCF applied to the error message created by participant 4 for task 1



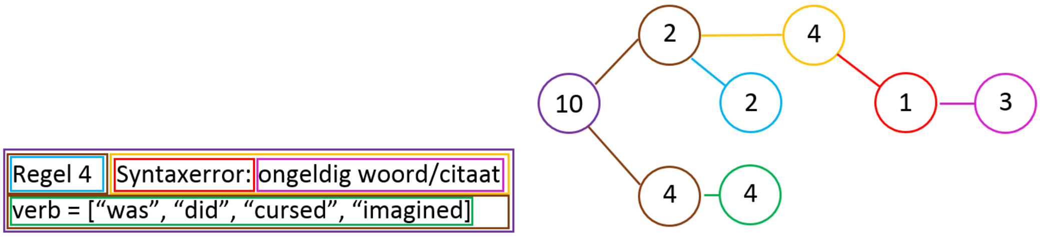
```
Regel 11 Syntaxerror: ongeldig getal   (1 11)
```

Figure E.22: EMCF applied to the error message created by participant 4 for task 2



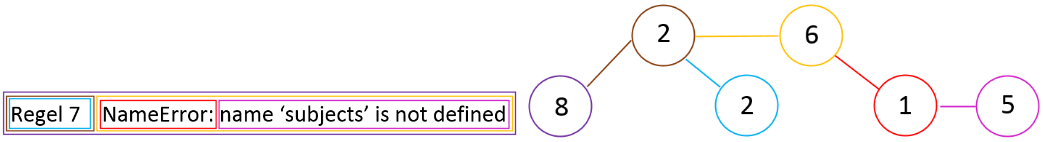
```
Regel 7 Syntaxerror: invalid syntax
              else
                ↑
```

Figure E.23: EMCF applied to the error message created by participant 4 for task 3



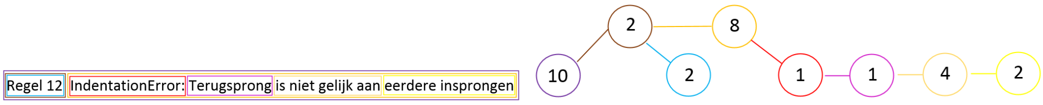
```
Regel 4 Syntaxerror: ongeldig woord/citaat
verb = [“was”, “did”, “cursed”, “imagined]
```

Figure E.24: EMCF applied to the error message created by participant 4 for task 4



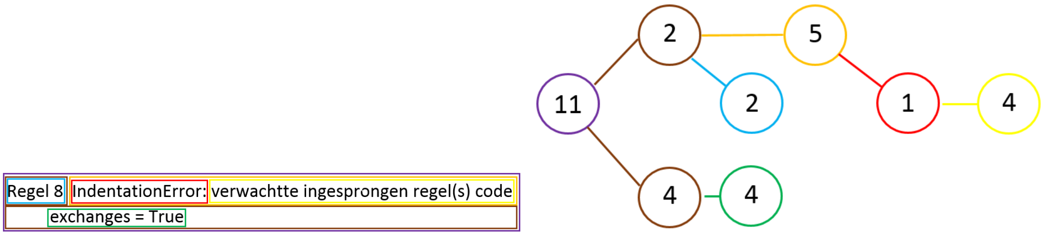
```
Regel 7 NameError: name ‘subjects’ is not defined
```

Figure E.25: EMCF applied to the error message created by participant 4 for task 6



```
Regel 12 IndentationError: Terugsprong is niet gelijk aan eerdere insprongen
```

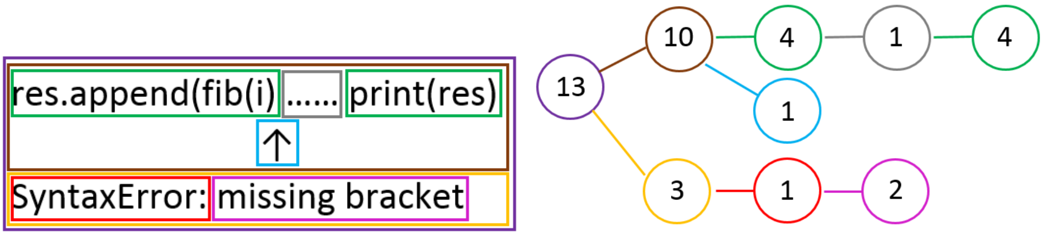
Figure E.26: EMCF applied to the error message created by participant 4 for task 7



```
Regel 8 IndentationError: verwachtte ingesprongen regel(s) code
        exchanges = True
```

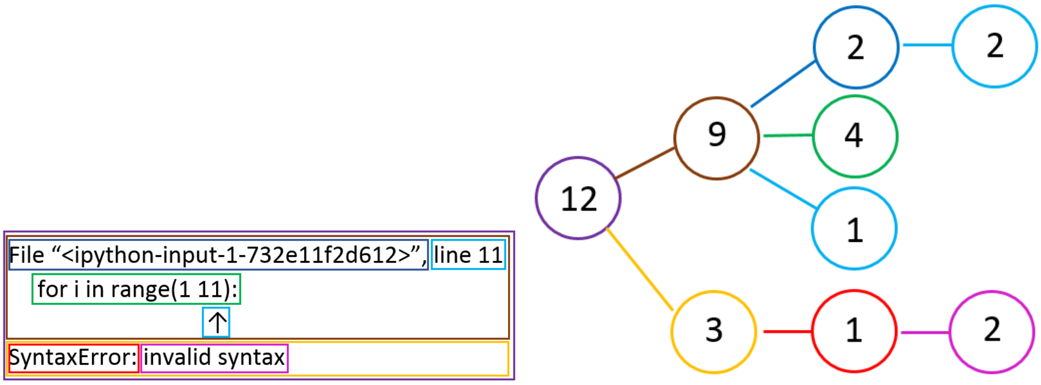
Figure E.27: EMCF applied to the error message created by participant 4 for task 8

E.5 Participant 5



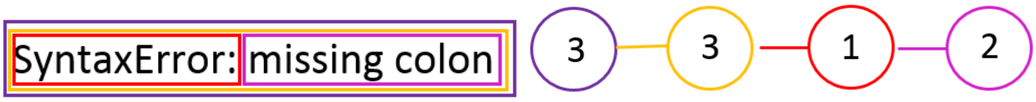
```
res.append(fib(i)...... print(res)
                ↑
SyntaxError: missing bracket
```

Figure E.28: EMCF applied to the error message created by participant 5 for task 1



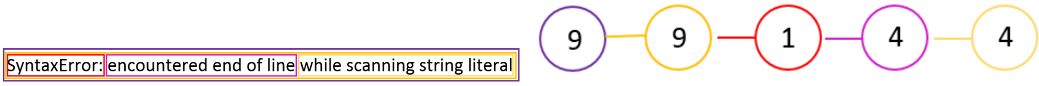
```
File “<ipython-input-1-732e11f2d612>”, line 11
    for i in range(1 11):
                    ↑
SyntaxError: invalid syntax
```

Figure E.29: EMCF applied to the error message created by participant 5 for task 2



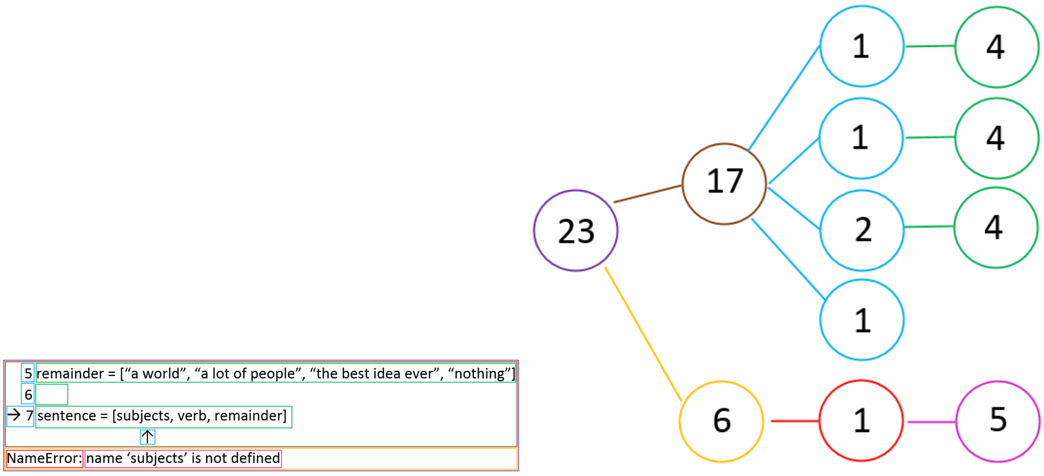
```
SyntaxError: missing colon
```

Figure E.30: EMCF applied to the error message created by participant 5 for task 3



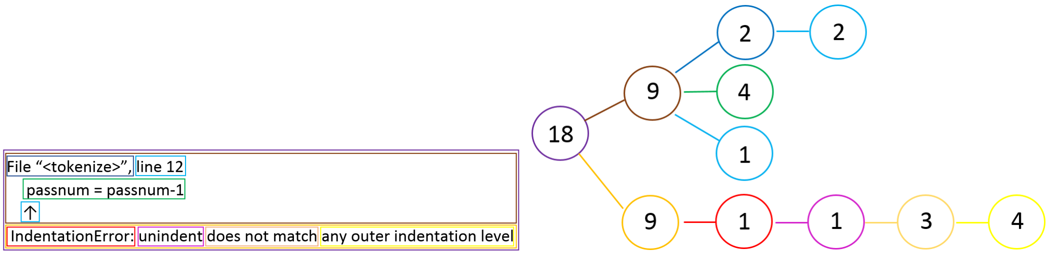
```
SyntaxError: encountered end of line while scanning string literal
```

Figure E.31: EMCF applied to the error message created by participant 5 for task 4



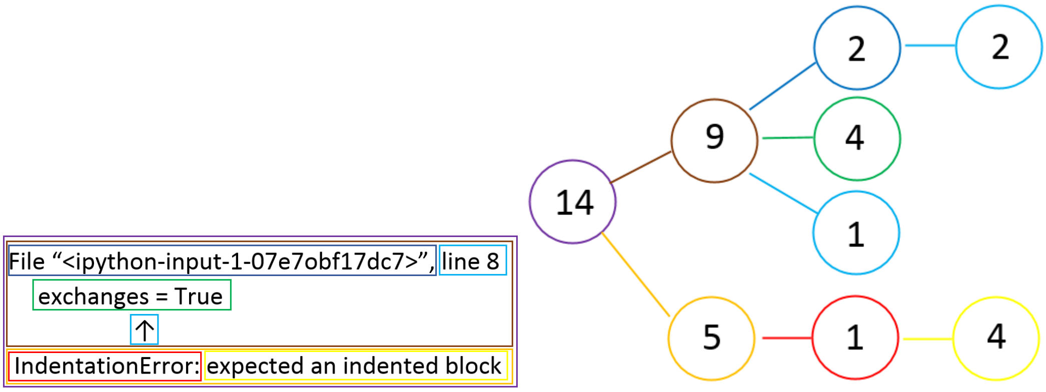
```
   5 remainder = [“a world”, “a lot of people”, “the best idea ever”, “nothing”]
   6
→ 7 sentence = [subjects, verb, remainder]
                    ↑
NameError: name ‘subjects’ is not defined
```

Figure E.32: EMCF applied to the error message created by participant 5 for task 6



```
File "<tokenize>", line 12
    passnum = passnum-1
    ↑
IndentationError: unindent does not match any outer indentation level
```

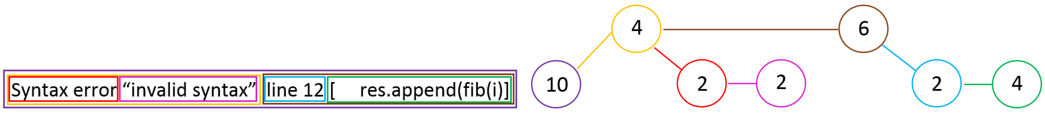
Figure E.33: EMCF applied to the error message created by participant 5 for task 7



```
File "<ipython-input-1-07e7obf17dc7>", line 8
    exchanges = True
              ↑
IndentationError: expected an indented block
```

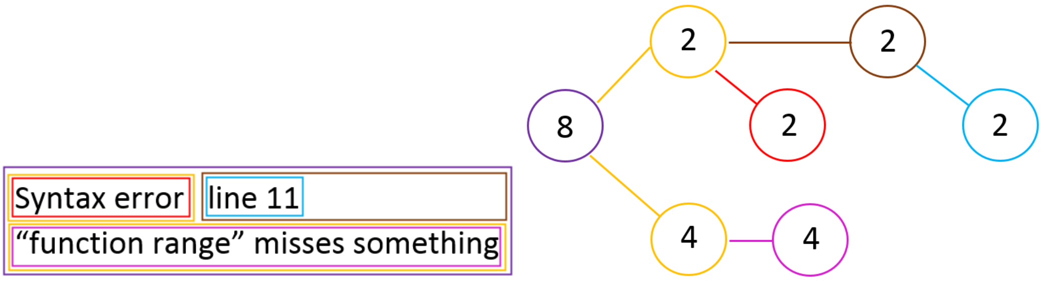
Figure E.34: EMCF applied to the error message created by participant 5 for task 8

E.6 Participant 6



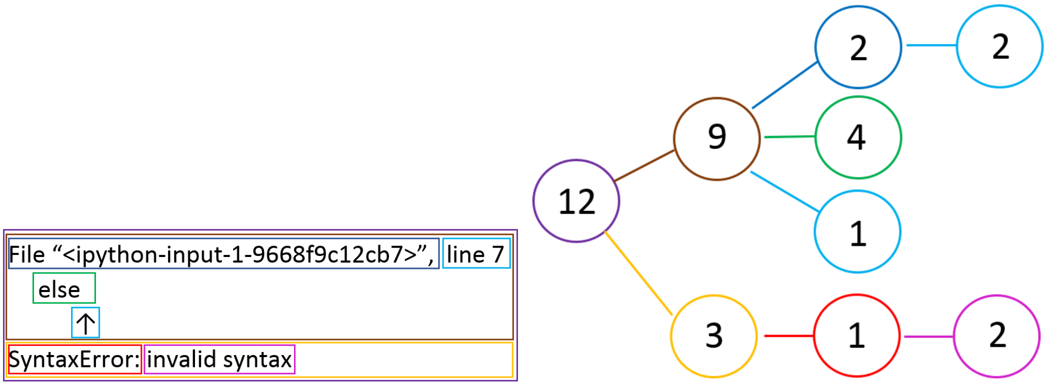
```
Syntax error "invalid syntax" line 12 [    res.append(fib(i)]
```

Figure E.35: EMCF applied to the error message created by participant 6 for task 1



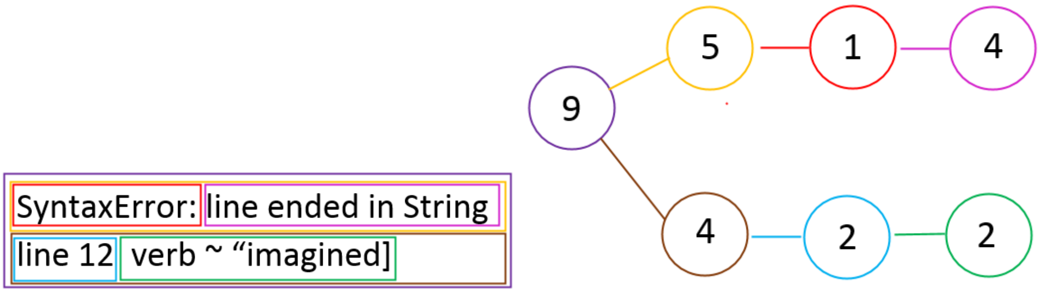
```
Syntax error line 11
"function range" misses something
```

Figure E.36: EMCF applied to the error message created by participant 6 for task 2



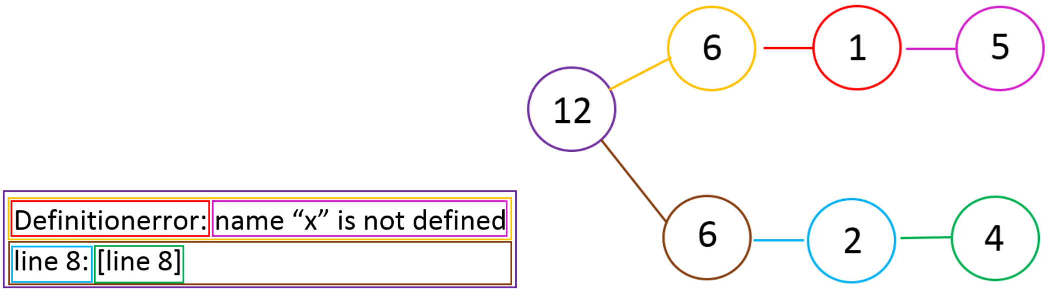
```
File "<ipython-input-1-9668f9c12cb7>", line 7
   else
       ↑
SyntaxError: invalid syntax
```

Figure E.37: EMCF applied to the error message created by participant 6 for task 3



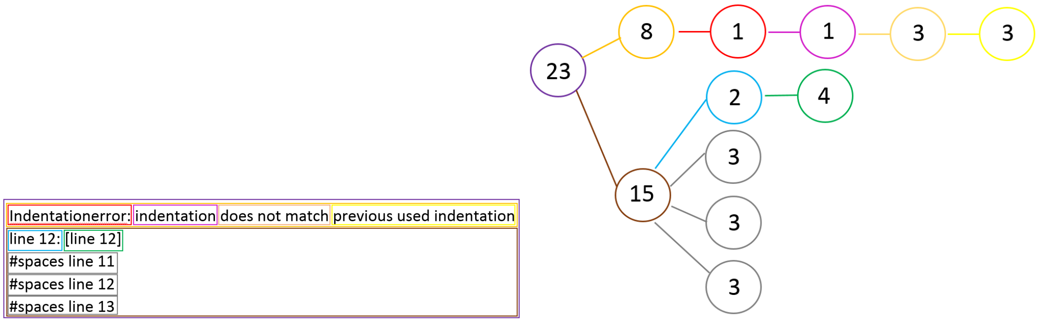
```
SyntaxError: line ended in String
line 12 verb ~ "imagined]
```

Figure E.38: EMCF applied to the error message created by participant 6 for task 4



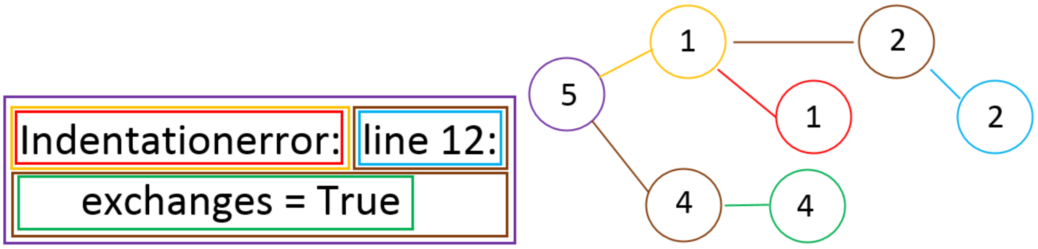
```
Definitionerror: name "x" is not defined
line 8: [line 8]
```

Figure E.39: EMCF applied to the error message created by participant 6 for task 6



```
Indentationerror: indentation does not match previous used indentation
line 12: [line 12]
#spaces line 11
#spaces line 12
#spaces line 13
```

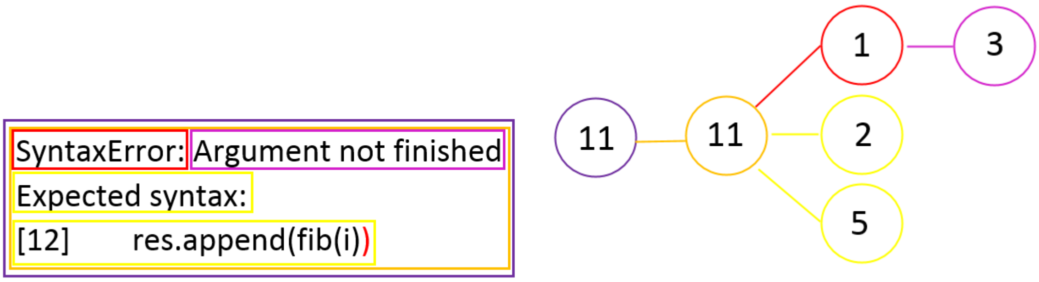
Figure E.40: EMCF applied to the error message created by participant 6 for task 7



```
Indentationerror: line 12:
    exchanges = True
```

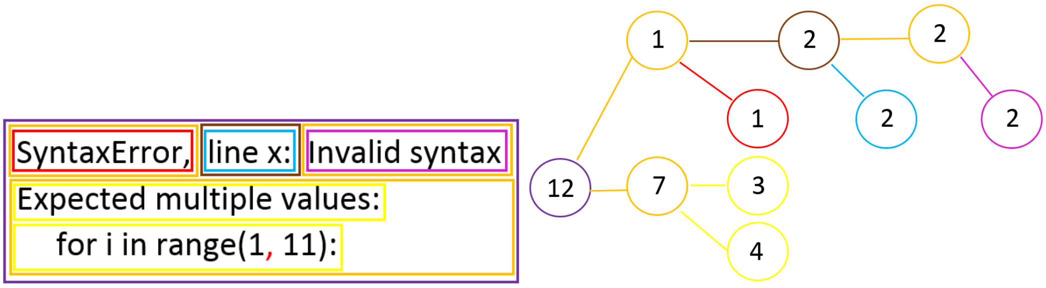
Figure E.41: EMCF applied to the error message created by participant 6 for task 8

E.7 Participant 7



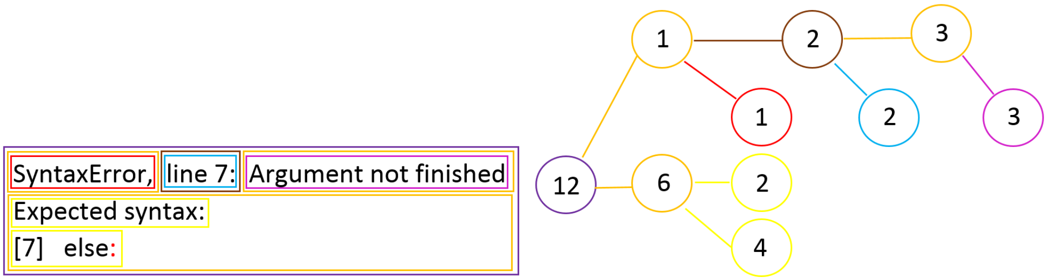
```
SyntaxError: Argument not finished
Expected syntax:
[12]      res.append(fib(i))
```

Figure E.42: EMCF applied to the error message created by participant 7 for task 1



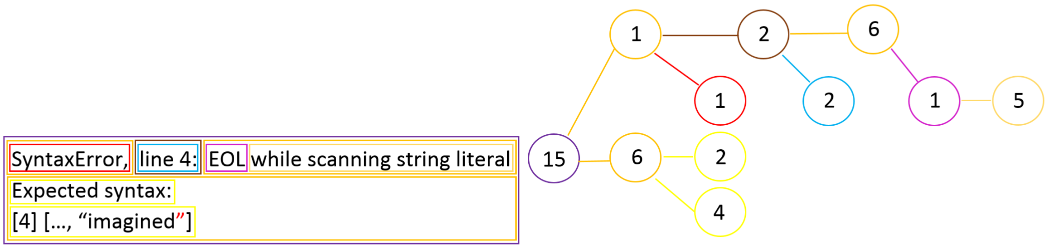
```
SyntaxError, line x: Invalid syntax
Expected multiple values:
    for i in range(1, 11):
```

Figure E.43: EMCF applied to the error message created by participant 7 for task 2



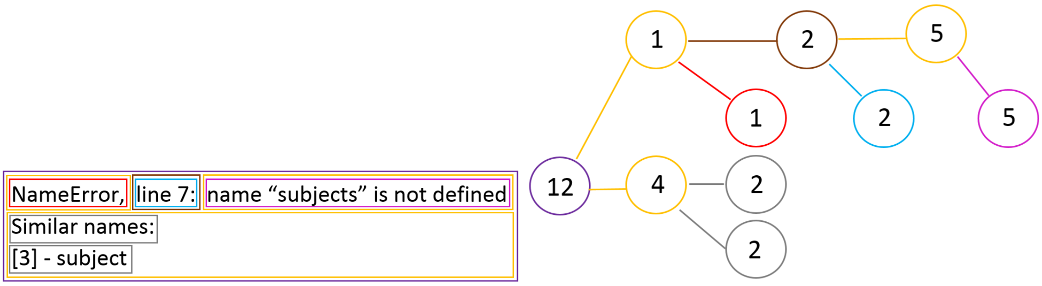
```
SyntaxError, line 7: Argument not finished
Expected syntax:
[7]   else:
```

Figure E.44: EMCF applied to the error message created by participant 7 for task 3



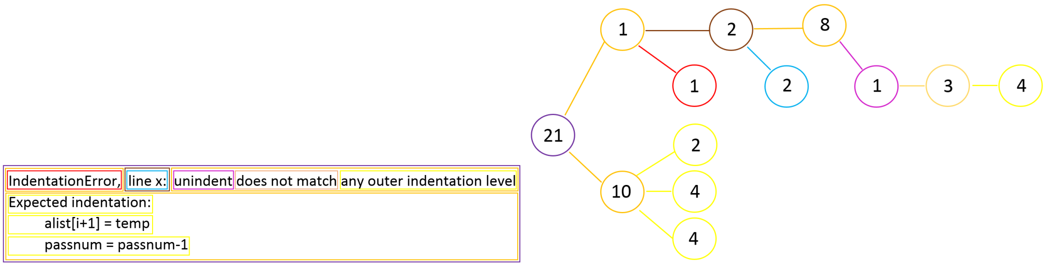
```
SyntaxError, line 4: EOL while scanning string literal
Expected syntax:
[4] [..., “imagined”]
```

Figure E.45: EMCF applied to the error message created by participant 7 for task 4



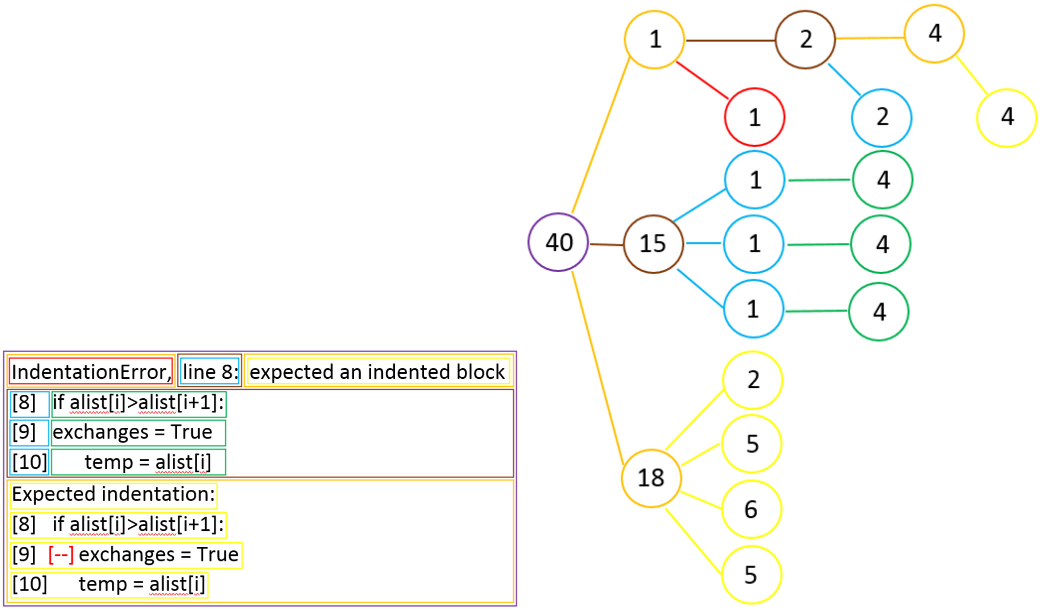
```
NameError, line 7: name “subjects” is not defined
Similar names:
[3] - subject
```

Figure E.46: EMCF applied to the error message created by participant 7 for task 6



```
IndentationError, line x: unindent does not match any outer indentation level
Expected indentation:
        alist[i+1] = temp
        passnum = passnum-1
```

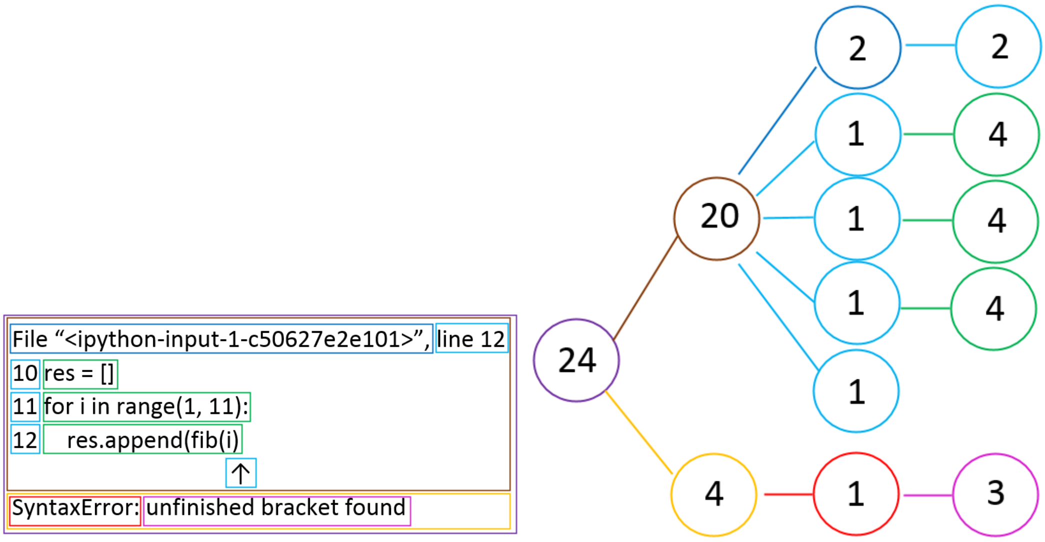
Figure E.47: EMCF applied to the error message created by participant 7 for task 7



```
IndentationError, line 8: expected an indented block
[8]   if alist[i]>alist[i+1]:
[9]   exchanges = True
[10]       temp = alist[i]
Expected indentation:
[8]   if alist[i]>alist[i+1]:
[9]   [--] exchanges = True
[10]      temp = alist[i]
```

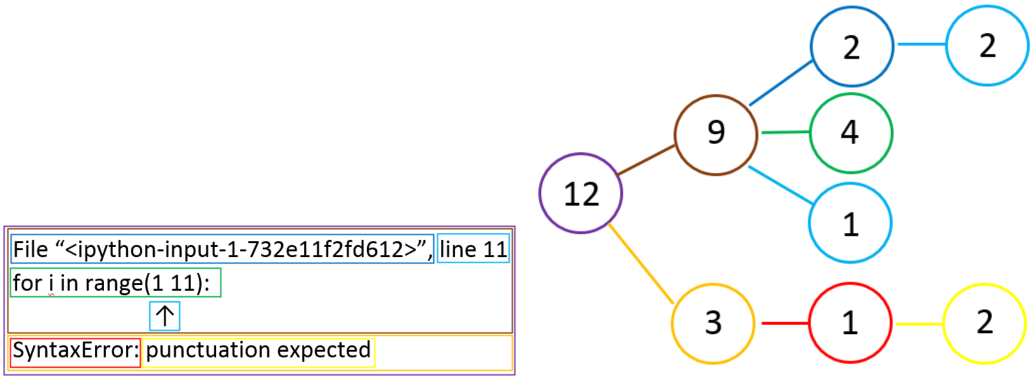
Figure E.48: EMCF applied to the error message created by participant 7 for task 8

E.8 Participant 8



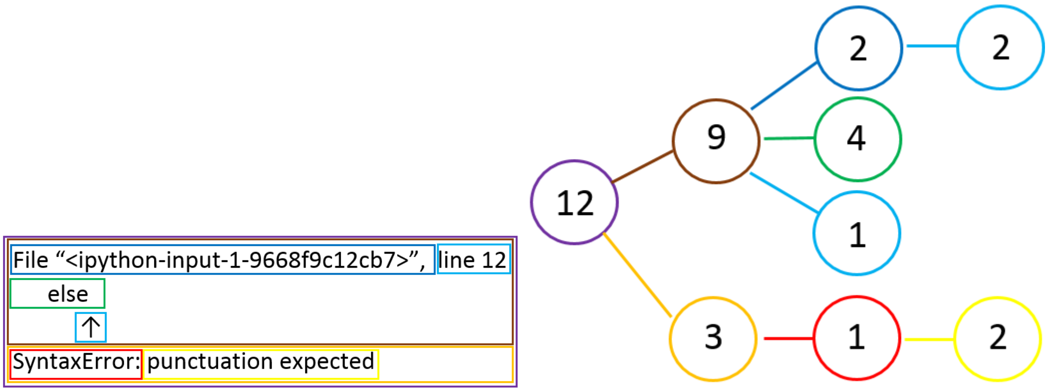
```
File “<ipython-input-1-c50627e2e101>”, line 12
10 res = []
11 for i in range(1, 11):
12     res.append(fib(i))
                         ↑
SyntaxError: unfinished bracket found
```

Figure E.49: EMCF applied to the error message created by participant 8 for task 1



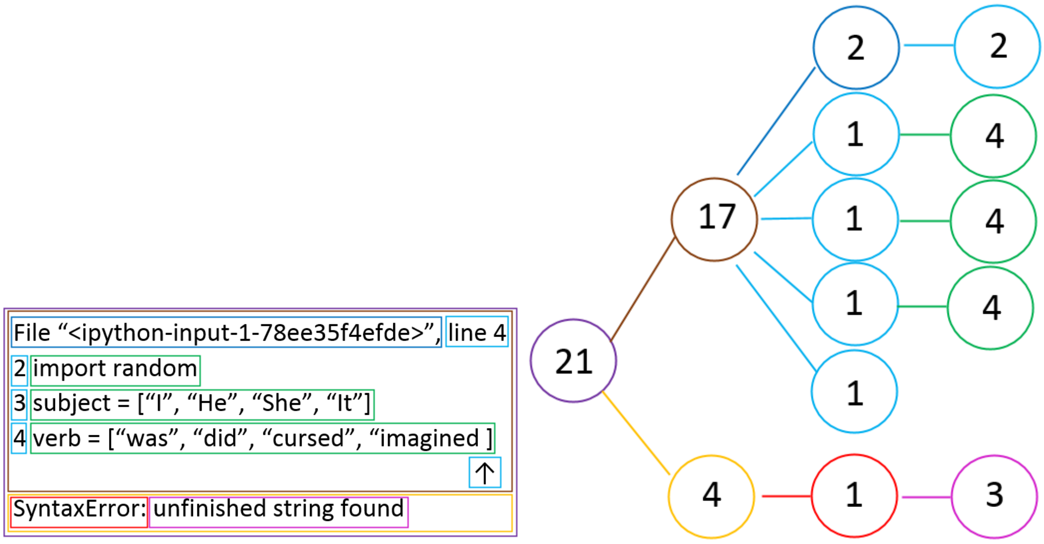
```
File “<ipython-input-1-732e11f2fd612>”, line 11
for i in range(1 11):
                 ↑
SyntaxError: punctuation expected
```

Figure E.50: EMCF applied to the error message created by participant 8 for task 2



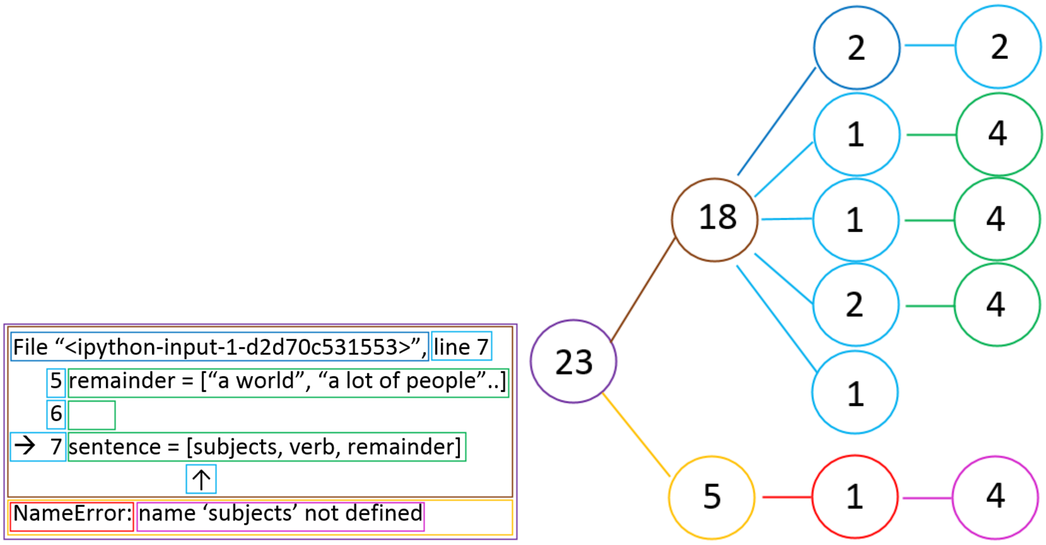
```
File “<ipython-input-1-9668f9c12cb7>”, line 12
     else
          ↑
SyntaxError: punctuation expected
```

Figure E.51: EMCF applied to the error message created by participant 8 for task 3



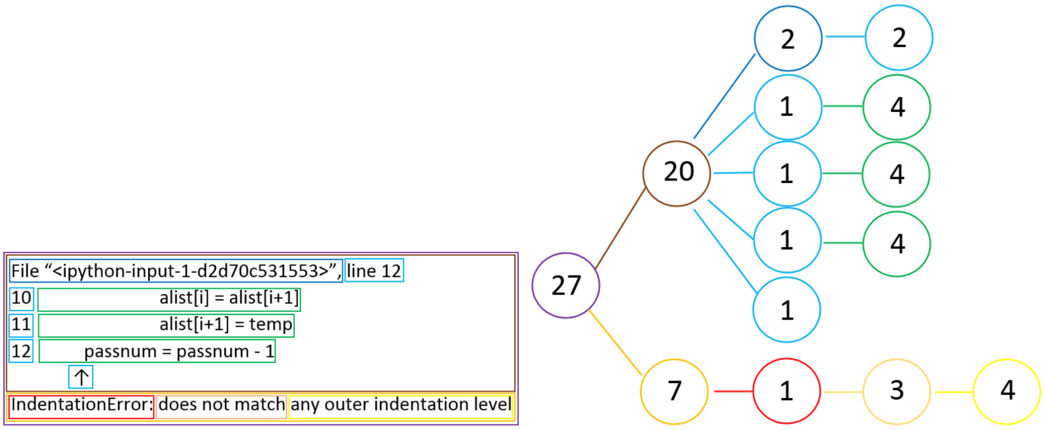
```
File “<ipython-input-1-78ee35f4efde>”, line 4
2 import random
3 subject = [“I”, “He”, “She”, “It”]
4 verb = [“was”, “did”, “cursed”, “imagined ]
                                             ↑
SyntaxError: unfinished string found
```

Figure E.52: EMCF applied to the error message created by participant 8 for task 4



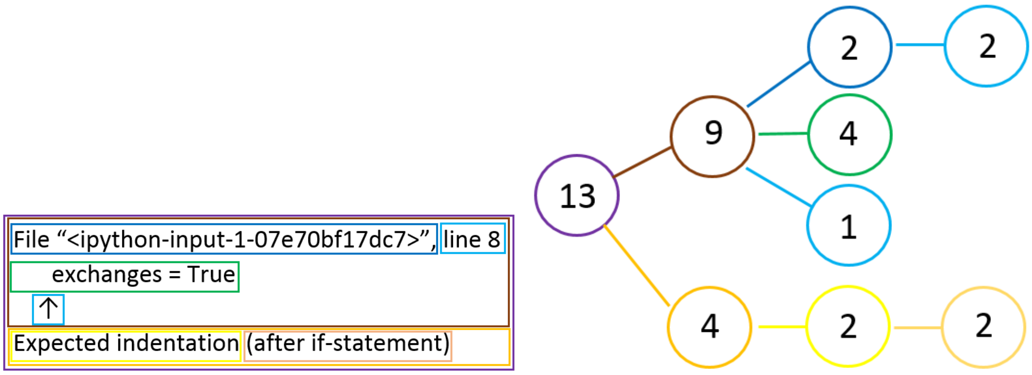
```
File "<ipython-input-1-d2d70c531553>", line 7
      5 remainder = ["a world", "a lot of people"..]
      6
→     7 sentence = [subjects, verb, remainder]
                  ↑
NameError: name 'subjects' not defined
```

Figure E.53: EMCF applied to the error message created by participant 8 for task 6



```
File "<ipython-input-1-d2d70c531553>", line 12
10                    alist[i] = alist[i+1]
11                    alist[i+1] = temp
12          passnum = passnum - 1
            ↑
IndentationError: does not match any outer indentation level
```

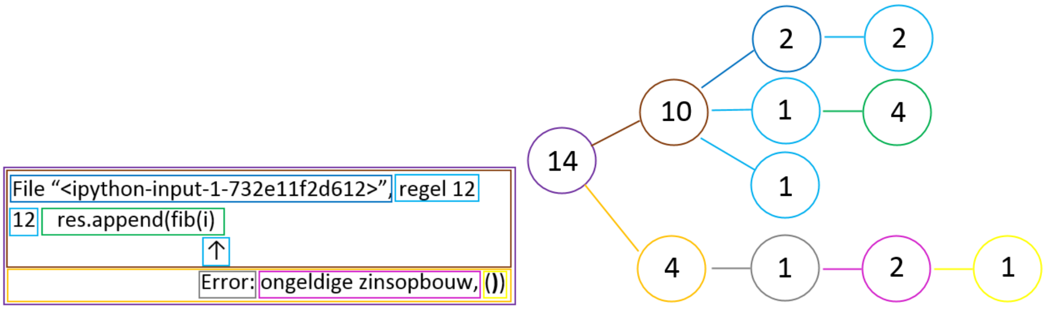
Figure E.54: EMCF applied to the error message created by participant 8 for task 7



```
File "<ipython-input-1-07e70bf17dc7>", line 8
    exchanges = True
  ↑
Expected indentation (after if-statement)
```

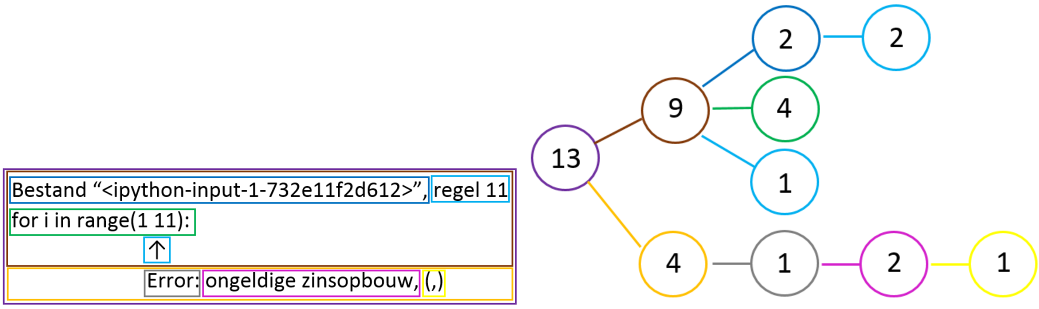
Figure E.55: EMCF applied to the error message created by participant 8 for task 8

E.9 Participant 9



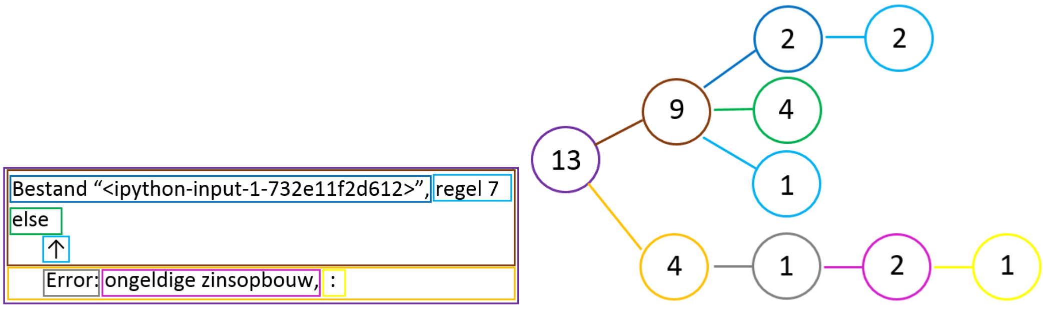
```
File "<ipython-input-1-732e11f2d612>", regel 12
12   res.append(fib(i)
                     ↑
                  Error: ongeldige zinsopbouw, ())
```

Figure E.56: EMCF applied to the error message created by participant 9 for task 1



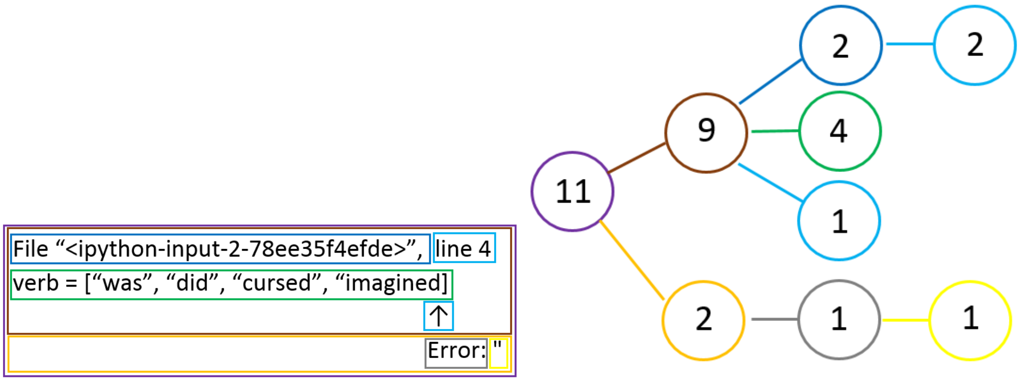
```
Bestand "<ipython-input-1-732e11f2d612>", regel 11
for i in range(1 11):
               ↑
           Error: ongeldige zinsopbouw, (,)
```

Figure E.57: EMCF applied to the error message created by participant 9 for task 2



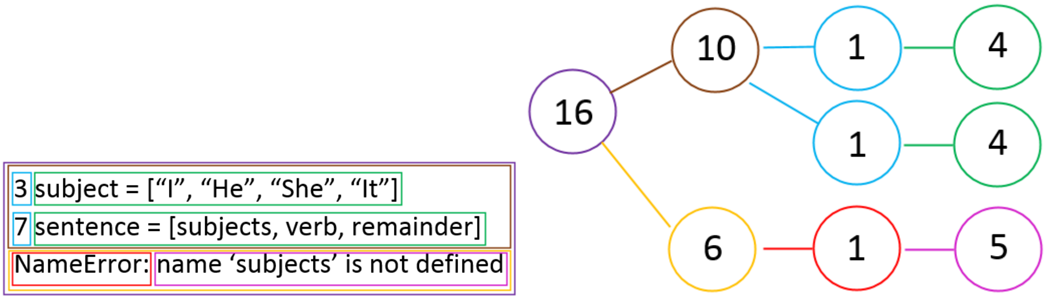
```
Bestand “<ipython-input-1-732e11f2d612>”, regel 7
else
   ↑
   Error: ongeldige zinsopbouw, :
```

Figure E.58: EMCF applied to the error message created by participant 9 for task 3



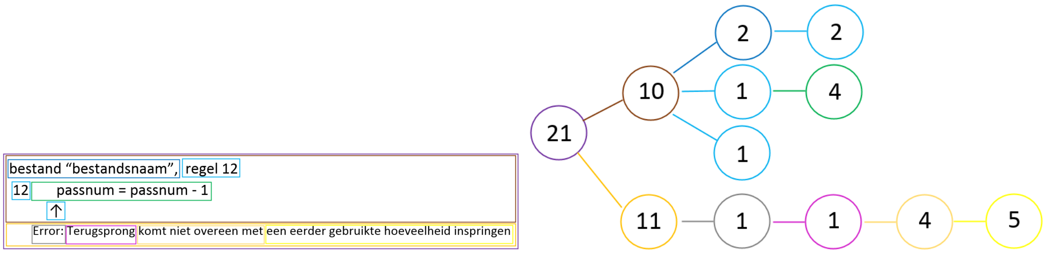
```
File “<ipython-input-2-78ee35f4efde>”, line 4
verb = [“was”, “did”, “cursed”, “imagined”]
                                          ↑
                                     Error: ''
```

Figure E.59: EMCF applied to the error message created by participant 9 for task 4



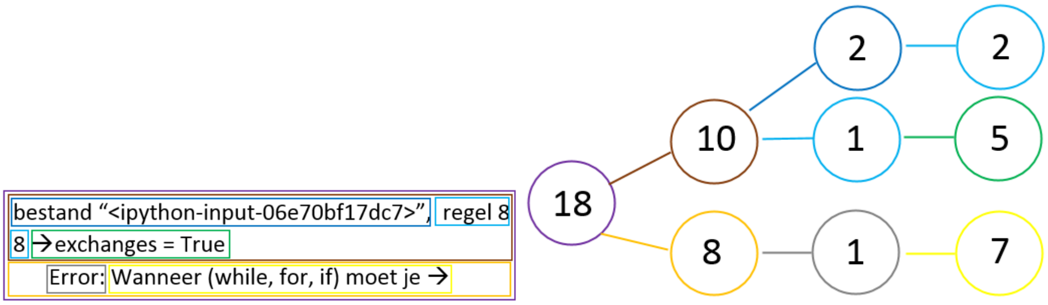
```
3 subject = [“I”, “He”, “She”, “It”]
7 sentence = [subjects, verb, remainder]
NameError: name ‘subjects’ is not defined
```

Figure E.60: EMCF applied to the error message created by participant 9 for task 6



```
bestand “bestandsnaam”, regel 12
12      passnum = passnum - 1
          ↑
    Error: Terugsprong komt niet overeen met een eerder gebruikte hoeveelheid inspringen
```

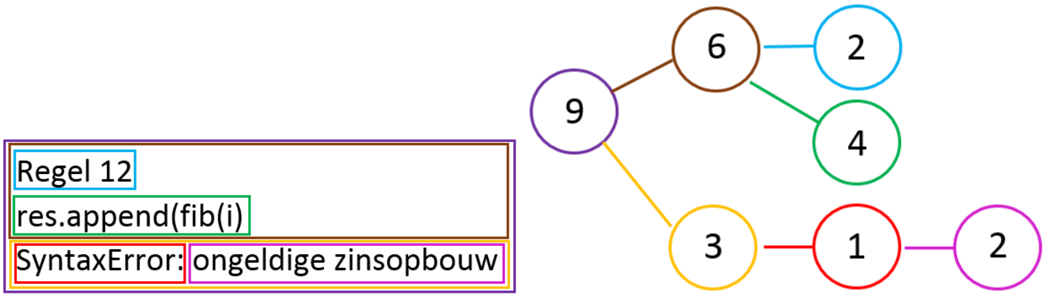
Figure E.61: EMCF applied to the error message created by participant 9 for task 7



```
bestand “<ipython-input-06e70bf17dc7>”, regel 8
8 →exchanges = True
    Error: Wanneer (while, for, if) moet je →
```

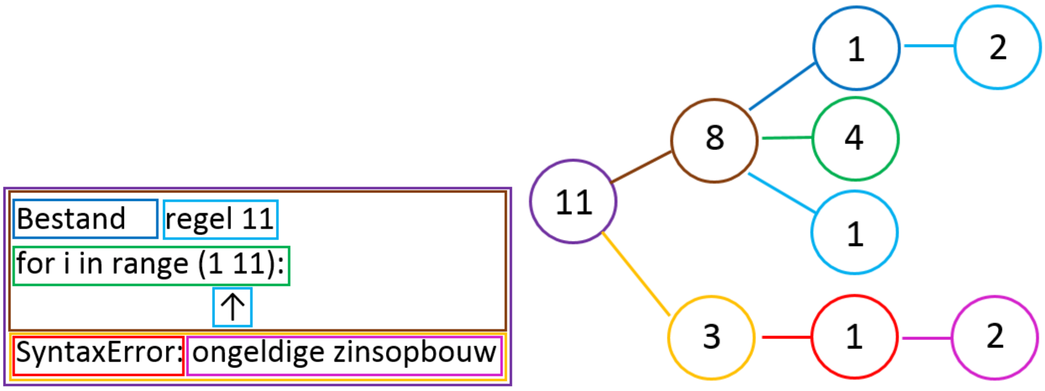
Figure E.62: EMCF applied to the error message created by participant 9 for task 8

E.10 Participant 10



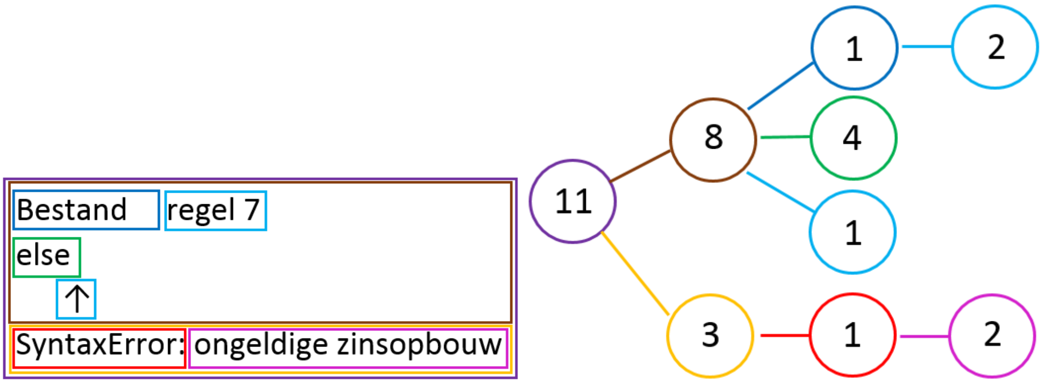
```
Regel 12
res.append(fib(i))
SyntaxError: ongeldige zinsopbouw
```

Figure E.63: EMCF applied to the error message created by participant 10 for task 1



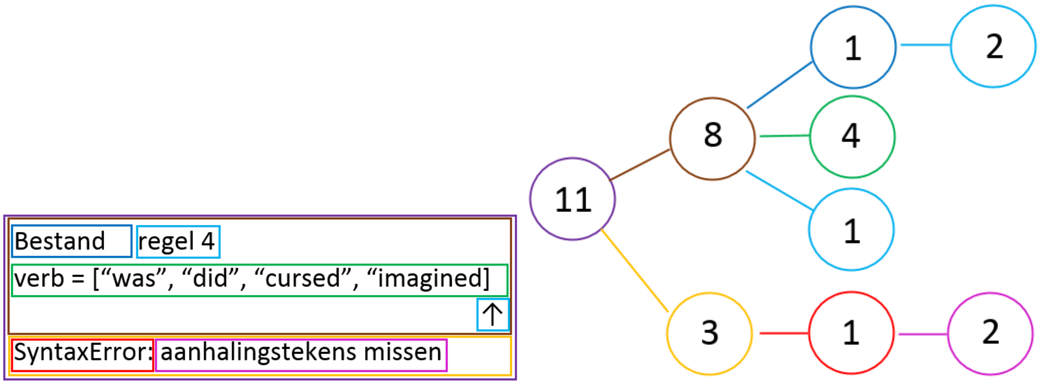
```
Bestand regel 11
for i in range (1 11):
                     ↑
SyntaxError: ongeldige zinsopbouw
```

Figure E.64: EMCF applied to the error message created by participant 10 for task 2



```
Bestand regel 7
else
     ↑
SyntaxError: ongeldige zinsopbouw
```

Figure E.65: EMCF applied to the error message created by participant 10 for task 3



```
Bestand regel 4
verb = [“was”, “did”, “cursed”, “imagined]
                                          ↑
SyntaxError: aanhalingstekens missen
```

Figure E.66: EMCF applied to the error message created by participant 10 for task 4

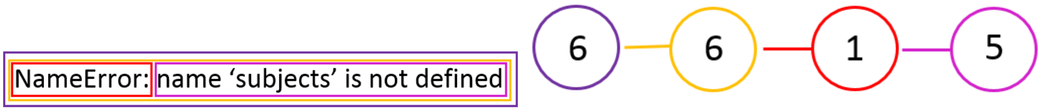


Figure E.67: EMCF applied to the error message created by participant 10 for task 6

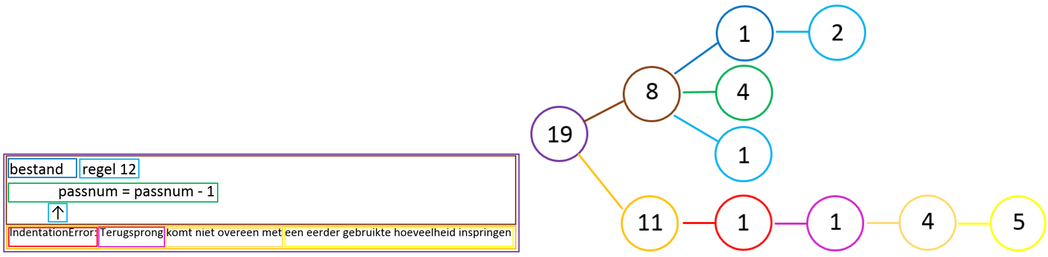

bestand regel 12
passnum = passnum - 1
↑
IndentationError: Terugsprong komt niet overeen met een eerder gebruikte hoeveelheid inspringen
19
8
1
2
4
1
11
1
1
4
5

Figure E.68: EMCF applied to the error message created by participant 10 for task 7

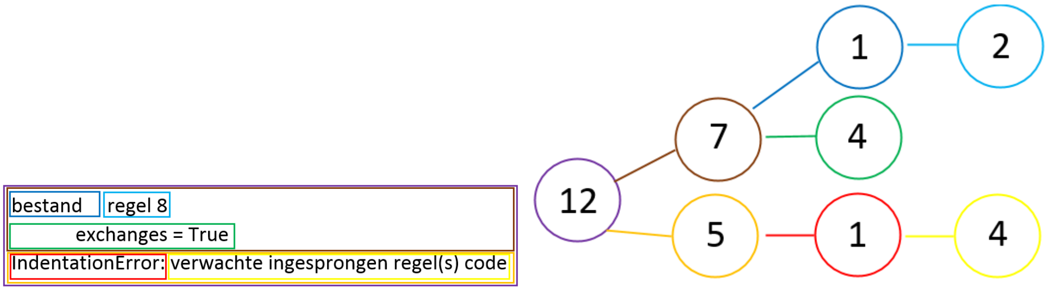


Figure E.69: EMCF applied to the error message created by participant 10 for task 8

E.11 Participant 11

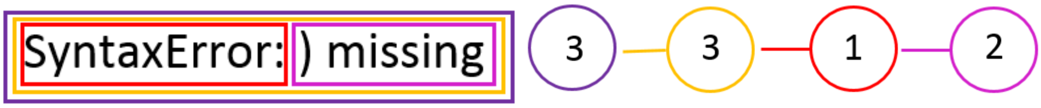


Figure E.70: EMCF applied to the error message created by participant 11 for task 1

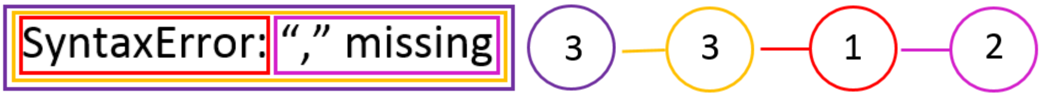


Figure E.71: EMCF applied to the error message created by participant 11 for task 2

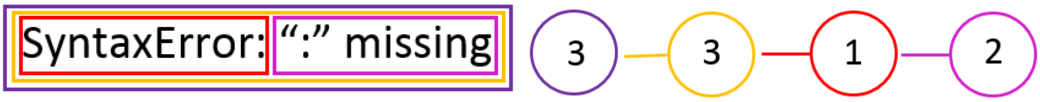


Figure E.72: EMCF applied to the error message created by participant 11 for task 3

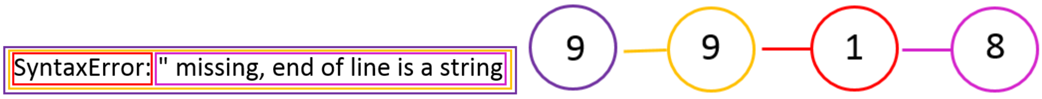


Figure E.73: EMCF applied to the error message created by participant 11 for task 4

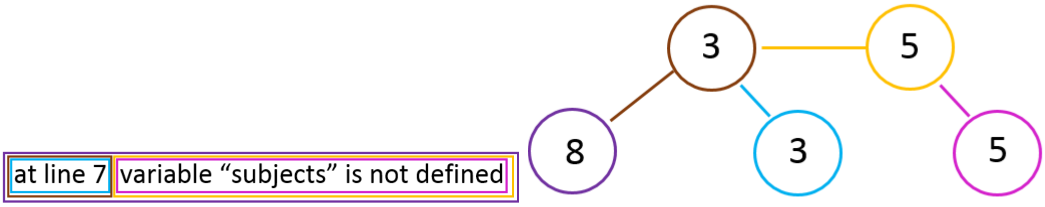


Figure E.74: EMCF applied to the error message created by participant 11 for task 6

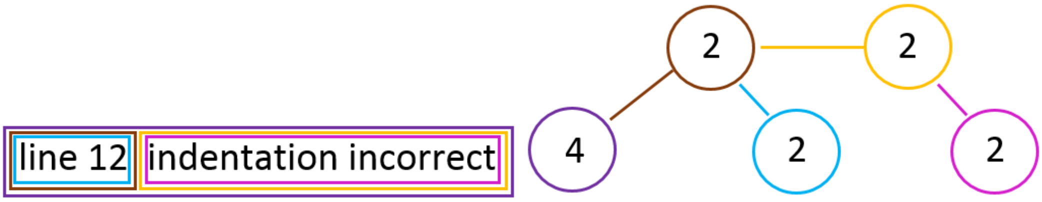


Figure E.75: EMCF applied to the error message created by participant 11 for task 7

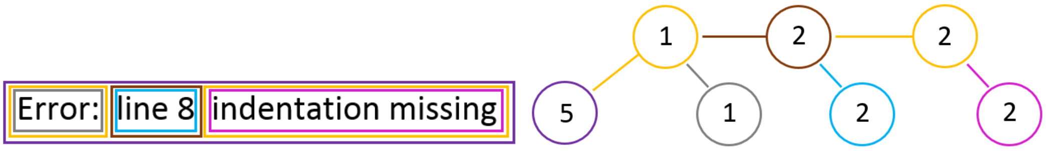


Figure E.76: EMCF applied to the error message created by participant 11 for task 8