DISS. ETH NO. 24076

Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

Tobias Kohn

Dipl. Math, ETH Zurich

born on January 31, 1982

citizen of Endingen (AG)

accepted on the recommendation of

Prof. Dr. Juraj Hromkovič, examiner

Prof. Dr. Bill Manaris, co-examiner

Prof. Dr. Thomas R. Gross, co-examiner

Prof. Dr. Jürg Gutknecht, co-examiner

2017

Für Sarah

Abstract

One of the great challenges in teaching to program is to help students understand how programming actually works. Students unavoidably develop misconceptions during their learning process, which must be corrected through skillful feedback. Such misconceptions reach from rather simple syntactical problems to deep underlying misunderstandings about what a computational machine is, and how it works. With the objective to assist the students in developing correct mental models, we must understand these misconceptions, and how to correct them.

An example of a misconception about the syntax is the well-documented problem that the assignment operator is believed to be symmetric, i. e., `2 = x` is seen as equivalent to `x = 2`. At the other end of the spectrum, we find fundamental misconceptions about the computational machine itself. One such misconception is to apply a model of mathematical procedure to the computational machine, and thereby attribute algebraic capabilities to the machine. This is evidenced by the belief that the statement `y = 2*x` establishes a link between the two variables x and y, so that any change of x is subsequently also reflected by y.

The primary and foremost means to detect misconceptions is through careful analysis of the students' mistakes and errors. While some errors lead to syntactically incorrect programs, which are not even accepted by the computer, and therefore quickly recognized, other errors show themselved in incorrect outputs once the program is run. However, the main question to answer is: what can we infer about a student's mental models and misconceptions from his or her errors?

This dissertation investigates the errors of novice programming students in high school. In particular, it provides evidence that some students apply a model of mathematical substitution when reasoning about variables in a program. This misconception is relevant as it directly pertains the sequential nature and data abstraction as employed in imperative computer programming; in fact, variables are a core concept in imperative programming. However, the dissertation also proposes a teaching method to specifically overcome this misconception by carefully addressing it in the classroom. Preliminary results are promising, and indicate that such dedicated teaching might indeed be effective.

During a period of five years, we have collected erroneous program of novice programming high school students. Based on this collection, the dissertation includes a list of common student errors. While there are several such collections of student errors already available, this

dissertation provides the first collection based on the programming language Python. Python has recently become a popular programming language for introductory courses. At the same time, it differs significantly from more traditional programming languages such as Java, in both syntax and execution model. Hence, many previously documented errors do not apply equally to Python.

Finally, the dissertation describes a parser for Python programs, capable of recognizing various error patterns, as documented in the collection of student errors. The parser has been implemented as part of the educational Python environment "TigerJython", which has already found widespread use. Moreover, with the possibility to discern different errors more accurately than what traditional Python environments provide, future research can study the relationships between the observed errors and the underlying misconception of students in more detail.

Zusammenfassung

Eine der grossen Herausforderungen im Programmierunterricht besteht darin, in den Lernenden das Verständnis zu fördern, wie Programmieren tatsächlich funktioniert. Während ihres Lernprozesses machen Schüler und Studenten unweigerlich Fehler und entwickeln Fehlvorstellungen, die durch gezielte Rückmeldungen korrigiert werden müssen. Diese Fehlvorstellungen reichen von einfachen syntaktischen Problemen bis hin zu einem tief liegenden Misverständnis darüber, was eine Rechenmaschine eigentlich ist und wie sie funktioniert. Mit dem Ziel, die Lernenden bei der Entwicklung korrekter mentaler Modelle zu unterstützen müssen wie ihre Fehler und Fehlvorstellungen untersuchen und studieren, wie sie korrigiert werden können.

Ein bekanntes Beispiel für eine syntaktische Fehlvorstellung liegt im Glauben, der Zuweisungsoperator sei symmetrisch. Die beiden Varianten `x = 2` und `2 = x` werden also als äquivalent angesehen. Auf der anderen Seite des Spektrums finden wir grundlegende Fehlvorstellungen über die Rechenmaschine selbst. Zum Beispiel tendieren einige Lernende dazu, ihr mathematisches Vorwissen auf die Rechenmaschine zu übertragen und gehen dann davon aus, dass die Maschine algebraische Fähigkeiten besitzt. Die Anweisung `y = 2*x` wird dann so verstanden, dass sie die beiden Variablen x und y fest miteinander verknüpft. Wenn wir später also den Wert von x ändern, dann würde diese Änderung auch sofort im Wert von y sichtbar.

Das wichtigste Mittel, um Fehlvorstellungen zu erkennen und zu untersuchen, ist eine sorgfältige Analyse der Fehler, die Schüler und Studenten machen. Während einige Fehler zu syntaktisch falschen Programmen führen, die schon vom Computer abgelehnt werden und daher einfach zu erkennen sind, äussern sich andere Fehler erst in den falschen Ausgaben, wenn das Programm ausgeführt wird. Die Hauptfrage ist aber in jedem Fall: Was für Schlüsse können wir aus den beobachteten Fehlern ziehen, um die mentalen Modelle und Fehlvorstellungen der Lernenden zu verstehen?

Die vorliegende Dissertation untersucht die Fehler von Schülerinnen und Schülern im Gymnasium, die einen Anfängerkurs in der Programmierung besuchen. Dabei präsentieren wir vor allem auch Indizien dafür, dass einige der Schüler ein Modell der mathematischen Substitution auf das Programmieren anwenden, wenn es darum geht, das Verhalten von Variablen im Programm zu erklären. Das ist insofern relevant, weil Variablen ein Kernkonzept der imperativen Programmierung darstellen. Tatsächlich hängen die Fehlvorstellungen der Schüler direkt mit

der sequentiellen Natur und der Datenabstraktion zusammen, zwei essentiellen Begriffen für die Programmierung. Die Dissertation stellt aber auch einen Ansatz vor, um diese Fehlvorstellungen in der Klasse zu korrigieren, indem das Problem bewusst und gezielt angegangen wird. Erste Ergebnisse dazu sind vielversprechend.

Während fünf Jahren haben wir eine Sammlung an fehlerhaften Programmen von Gymnasialschülern angelegt. Basierend auf dieser Sammlung enthält die Dissertation auch eine Liste mit häufigeren Fehlern. Natürlich gibt es bereits einige solche Listen und Zusammenstellungen, allerdings präsentieren wir die erste Liste zur Programmiersprache Python. Python erfreut sich zur Zeit wachsender Beliebtheit als Anfängersprache. Gleichzeitig unterscheidet sich Python aber auch relativ stark von traditionelleren Sprachen wie Java, und zwar sowohl in der Syntax als auch im Ausführungsmodell. Entsprechend lassen sich viele frühere Ergebnisse nicht einfach übertragen.

Schliesslich stellt diese Dissertation auch einen Parser für Python vor, der eine ganze Reihe von Fehlern erkennt, die zuvor dokumentiert wurden. Dieser Parser ist bereits Teil der Python-Umgebung “TigerJython” für den Unterricht, die relativ weit verbreitet ist und vielfältig eingesetzt wird. Darüber hinaus ermöglicht die systematische Erkennung von Fehlern aber ach zukünftige Forschung, insbesondere das Studium der Beziehungen zwischen den beobachteten Fehlern und den tatsächlich zugrunde liegenden Fehlvorstellungen.

Acknowledgments

Apart from the scientific results, the writing of this dissertation has brought me in contact with many great people. I feel a deep gratitude for all the opportunities, the support, and the friendship I have received during this time, and I would like to express my highest appreciation. In particular, I would like to thank the following people.

When I first approached Profs. Juraj Hromkovič and Jürg Gutknecht with my ideas for this dissertation, I quickly found their full endorsement. I am most grateful for their extensive support, their trust and skillful advice. Both allowed me to always pursue my own ideas, and learn from my own mistakes, while still guiding and assisting me when necessary. In addition, Prof. Thomas R. Gross has supported and guided me in my studies of parsing technologies. Moreover, he provided very helpful, extensive and detailed feedback to my manuscripts of this dissertation, for which I am most grateful.

I further had the extraordinary fortune of also profiting highly from the valuable insights and feedback of Prof. Bill Manaris during our numerous long conversations. In addition to his great advice on many aspects of my work, he helped me connect to the community of Computer Science education, and invited me to join him at the SIGCSE conferences. I highly appreciate his support.

I would like to extend my deepest gratitude to the team at the chair of information technology and education. In particular, Giovanni Serafini, Dennis Komm, Hans-Joachim Böckenhauer, Fabian Frei, Urs Hauser, Björn Steffen, Lucia Keller, Ivana Kosírova, Elizabeta Cavar, Jan Lichtensteiger, and Blanca Höhn, whose support has proven invaluable on so many occasions. As I have had the pleasure of working closely together with Giovanni Serafini and Dennis Komm on several Workshops, I feel particularly connected to them, and enjoyed our most productive collaborations. In addition, Hans-Joachim Böckenhauer has amazed me with his uncanny ability to spot (typing) errors in my manuscripts. I would like to thank him and Dennis Komm for their careful proofreading, and their skilled technical feedback.

Special thanks go to Blanca Höhn and Denise Spicher, who have been most patient with me, and whose support and assistance in all the administrate matters has been very valuable.

As Urs Hauser and Ivana Kosírova both teach programming at high schools themselves, they offered me the opportunity to test many ideas at their schools as well, and their professional

feedback on my work has always been very encouraging and helpful.

When I initially started to develop a small educational Python environment, Prof. Aegidius Plüss quickly saw the potential of reaching out to a larger community. Meanwhile, the environment has grown into the much larger *TigerJython*-project, in particular thanks to his incessant and enthusiastic work. The TigerJython-project has received various funds, has been translated to several languages, and spread far beyond Switzerland. Aegidius Plüss has not only provided invaluable feedback to all aspects of my parser and programming environment, but also always given great advice.

Lorenz Halbeisen of the Mathematics department has been my mentor in mathematical logic. He has guided me in developping a much deeper understanding of mathematics, and thereby enabled me to see much more clearly the differences between mathematics and programming. I highly appreciate his help and our insightful discussions.

I would like to thank Martin Zimmermann. During my entire doctoral studies, I have been teaching at his high school. As principal, he has always generously supported my studies and projects without hesitation.

Most importantly, I feel a deep gratitude towards Sarah Frei, who has always given me her full support and love.

Contents

Chapter 1

Introduction

1.1 Introduction

Making errors is an important and integral part of every learning process. By committing errors and receiving feedback, we learn to modify and improve our conceptions and understanding, and discern correct ideas and approaches.

As educators, we want to provide the students with the necessary feedback, and foster them in this process of learning. But how do we give them meaningful feedback? And what do we actually know about the students' thinking, reasoning and their conceptions of our field?

This dissertation investigates the errors students make when learning to program in Python. We claim that some errors are due to fundamental misconceptions about how programming works in general. In particular, some high school students apply their mathematical preknowledge to programming, even when the used concepts strongly differ from mathematics to programming. However, we provide evidence that the students profit from directed teaching that addresses such misconceptions.

On the other side of the spectrum, some errors can be attributed to simple (syntactical) mistakes when writing a program. Such mistakes are usually easily fixed, once they are spotted. If we assume that the novice programmer making the mistake actually knows how to fix it, a simple error messages at the correct position might suffice.

Yet, can a Python compiler do more than point out the position of simple and obvious syntax errors? Might it even be able to recognize common error patterns, and assist us in classifying errors made by novice programmers? Could an elaborate system, in the end, provide helpful feedback to a significant part of students?

We claim that a Python compiler actually can recognize a large class of error patterns that are found in novice programmers' code. To support our claim, we collected various errors made by our high school students, and built a parser for Python to recognize these errors. During this process, we also discovered some limitations to what a parser can achieve.

Research question. Assume, we want to teach an introductory programming course in Python. What errors and misconceptions might the students in our course have? What kind of feedback could the Python system give to the students, so as to assist them in developing correct mental models of Python programming? And what support and feedback do we, as educators, need to provide to the students to foster their learning?

Methodology. In the course of this dissertation, we have collected Python programs of high school students, who were learning to program, over a period of about five years. We then searched for common error patterns in these programs, and tried to identify the causes and sources of the errors.

The entire research is based on relatively small sample sizes. However, our objective is to understand the errors made by the students, and find relationships between the students' thinking and their errors and mistakes. In other words, we focus mostly on individual errors or problems, its causes and sources, and are rarely concerned with the frequencies of their occurrence. In particular, this is not a statistical research study, and the evidence we provide is more anecdotal than supported by extensive statistics.

Python

In the course of the past few years, Python has become a popular choice as educational programming language (see, e. g., Guo [18], or Grandell et al. [15]). At the same time, we notice that little research has been done that specifically addresses Python. In particular, recent studies about syntax errors and novices' difficulties have usually been done in Java, C++, or Scheme (see Section 6.2).

Since Python's syntax and execution model differs greatly from the languages otherwise studied, not all results can easily be applied or translated to Python. Moreover, current implementations of Python provide only very crude error messages, which do not discern between different kinds of syntax errors. This makes an analysis of the most frequent novice errors, for instance, difficult or even impossible. As a result, we simply do not know how common various syntax errors in Python really are.

Hence, a first step towards comprehensive research about Python in education must be to establish a set of (syntactical) errors that are recognized by compilers and allow reproducible and comparable results.

An Example of a Student Error

We give an example of a student program from our data set with a brief discussion of the errors contained. Even though this particular program has not been part of this thesis' studies, the

reader gets a first-hand impression of our methodology, and what errors might be found in a student's program.

Consider a simple question, asking the student to write a program to solve the following problem (slightly abbreviated and simplified): *"Find all Pythagorean quadruples* (a, b, c, d)*, fulfilling the equation* $a^2 + b^2 + c^2 = d^2$*, where* $a, b, c \leq 100$*. All four numbers* a, b, c, d *are natural numbers. An example of such a quadruple is* $(1, 2, 2, 3)$*."*

The student's answer. The answer of one student is given below – a program that is clearly incorrect in various ways (note that the student wrote the answer on a sheet of paper without access to a computer).

```
a = 0
repeat 100:
    b = a+1
    c = a+1
    d = a+2
    if d*d = a*a+b*b+c*c and a, b, c <= 100:
        print a,b,c,d
    a += 1
```

At first sight, the student seems to have not understood how to properly iterate over several variables using nested loops. The three variables b, c and d are not iterated over independently, but rather depend on the variable a. The only quadruple this program is ever to find is $(1, 2, 2, 3)$. This is actually the very quadruple that was given as an example in the problem itself!

Given that the problem contained $(1, 2, 2, 3)$ as an example, we might surmise that the student took it to be a pattern for all quadruples to be found. Thus, the student's answer does actually not support any conclusions about his or her ability to write a program with nested loops.

In addition to the overall algorithmic idea behind the program, there are two syntax errors in line 6 after the `if`. First, a comparison needs a double equal sign == instead of the assignment operator =. This is probably a simple typing error. Second, the comparison `a, b, c <= 100` is not syntactically valid in Python, either. It is, again, probably taken from the problem's question that stated: $a, b, c \leq 100$.

This last error is an example of students transferring mathematical notation directly to programming. It actually seems that students only have a «fragile» understanding of Python's syntax, failing them in some subtler cases (cf. [35]). They might just infer from the valid comparison `a <= 100` that `a, b, c <= 100` is close enough to be valid as well.

Conclusion from the example. The upshot of this example is that students' errors have various different causes. Not all errors truly reveal a misconception or inability of the student. The incorrectness of the above program, for instance, is more likely the cause of a misunderstanding of the problem asked.

On the other hand, even mere syntax errors can have different causes. Whereas some syntax errors are in actuality just simple mistakes and typing errors, other errors reveal misconceptions about Python's syntax. In some cases, the novice programmer might just not know how to correctly implement his or her plan.

When designing a Python environment to provide feedback to the novice programmer, the biggest challenge we face is that each student might require another type of error message, so as to be helpful to him or her. If, for instance, the student, in the example above, knows that a comparison is written with a double equal sign, a minimalistic error message at the correct position will suffice. In contrast, in the case of the comparison $a, b, c \leq 100$, the error message might have to be much more tailored to the problem at hand to be of any help at all. Just telling the student that the comma after the a is invalid might imply to the student that Python uses a different separator symbol, e. g., a semicolon. Even with a corrected syntax, i. e., `(a, b, c) <= 100`, the comparison is still invalid. Accordingly, the error message should be careful not to suggest a putative solution that turns out to be incorrect as well.

1.2 Theses and Contributions

We propose three theses, for which we will provide supporting evidence in the course of this dissertation. Due to the nature of the theses, we cannot prove them in a strict sense. Instead, we present concrete artifacts from students' programs, possible interpretations of these artifacts, and eventually show that our claims are consistent with previous research.

Thesis 1. Some common misconceptions and errors made by novice programming students can be explained as the students applying a mathematical model of syntactical substitution to program execution.

Thesis 2. It is possible to directly improve the students' understanding and cognitive concept of variables, and the computational model through explicit teaching.

Thesis 3. The parser can correctly identify and report at least 75 % of syntax errors that are made by high school novice programmers in Python.

Contributions

TigerJython. As part of this dissertation, the author has written an educational development environment for Python programming, called *TigerJython*. In the context of the dissertation, we particularly focus on the *parser*, that has become part of this environment (Chapter 7).

TigerJython builds on Jython [28], a Python implementation that runs on the Java platform. On top of Jython, we use a customized parser as described in Chapter 7. It also includes a debugger and visualizing system, inspired by Philip Guo's *Python Tutor* [19], and provides a simple student-oriented interface. *TigerJython* has been freely available under [29]. More details are given in Chapter 2.

This dedicated Python environment allowed us to try different ideas for the parser directly in class, and hence gain experience from classroom use. The programs collected for our study in Chapter 8 have also been collected through *TigerJython*.

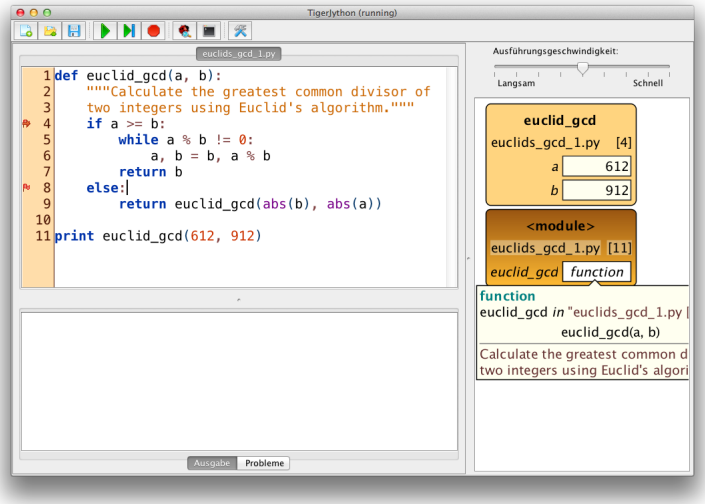


Figure 1.1: A screenshot of the educational Python environment *TigerJython*. The program is currently running, and the debugger, inspired by P. Guo's Python Tutor, shows on the right.

Apart from its direct use as part of the dissertation, *TigerJython* has found widespread use, particularly in Switzerland and Germany. The website analytics for [29], as presented in Fig. 1.2, show downloads of more than 3000 copies of *TigerJython* during August 2016 (after the summer holidays). Some high schools, however, have installed a single copy of *TigerJython* to be used by the entire school. The real number of actual users is thus difficult to estimate. Yet, the statistics in Fig. 1.2 also shows how often an instance of TigerJython asks the server whether an update is available. This gives a conservative estimate of more than 750 users each weekday.

In Switzerland, the programming environment *TigerJython* spawned a project comprising several tutorials and workshops, most notably the extensive online tutorial "Programming Concepts" [3]. While this project is clearly not part of the dissertation, its use of the *TigerJython*-environment can be seen as additional proof of concept for the idea of an educational Python environment. The TigerJython-tutorial received multiple funds and has been translated from German to English and French.

	Aug	Sep	Oct	Nov
Application downloads	3 484	1 492	794	1 379
Checking for updates	10 631	17 670	8 377	16 242

Figure 1.2: Download statistics for *TigerJython* during four months in 2016. Note that the drop in October is probably due to about two weeks of Swiss national holidays.

The environment *TigerJython* has also been used as the basis for a second Python environment *JEM* (Jython Environment for Music) [38], aimed at teaching Python programming by making music. This teaching program and Python environment has been presented at two workshops titled "Making Music with Computers: Creative Programming in Python", being held at SIGCSE technical symposiums 2015 and 2016 in Kansas City and Memphis, respectively.

Finally, TigerJython is also used in the *WEKA*-suite for Data Mining and Machine Learning of the university of Waikato, New Zealand [49].

Concepts of variables. The contents of Chapters 4 and 5 are to be published at the upcoming SIGCSE technical symposium 2017 in Seattle. The paper "*Variable Evaluation: An Exploration of Novice Programmers' Understanding and Common Misconceptions*" by the author of this dissertation has been accepted, and will be presented in March 2017 at the conference.

Classification of errors. Current implementations of Python provide only simple error messages and do not (or only in a very limited fashion) discern between various kinds of syntax errors. In order to study the errors made by novice programmers, for instance, we require more precise error messages. In particular, a statistical study is hardly possible without automated assessment of the objects to study (in this case the errors).

With our classification of syntax errors in Python we provide a basis for such studies. Moreover, the parser presented in this dissertation is an already working implementation.

1.3 Organization of the Dissertation

The development environment *TigerJython* is briefly presented in Chapter 2.

Chapter 3 provides an overview of Python's features and grammar. We focus on those aspects of Python which are most relevant for the dissertation, in particular the parser (Chapter 7). Appendix 9.2 contains a complete copy of Python's grammar.

Chapters 4 and 5 discuss the concepts of variables in mathematics and programming. We argue that some students apply the mathematical model to programming, resulting in misconceptions about how programming actually works. In particular, we discuss Theses 1 and 2.

Chapters 6, 7, and 8 present the errors we have collected from our students' programs, as well as the parser we have built. This part also contains an analysis of the parser's performance concerning error recognition, as well as a discussion of Thesis 3.

Appendix 9.2 gives a complete list of all the errors our parser can currently recognize.

Chapter 2

The Educational Python-Environment TigerJython

2.1 Introduction

As part of this dissertation, we have written a development environment for Python programming. This environment is called *TigerJython* and addresses specifically novice programmers, in particular (high school) students. While the remainder of this dissertation focuses on an essential aspect of TigerJython, namely the parser, this section provides a more comprehensive overview of the development environment, and the rationales behind its design.

Assisting the novice programmer through software. According to du Boulay, programming novices face five major areas of difficulties when learning to program [13]:

1. **Goals and Objectives.** What is programming for, and what problems can be solved through programming?
2. **The Notional Machine.** What are the properties, capabilities and limitations of the machine that executes the program?
3. **Syntax and Semantics.** Which rules and basic vocabulary build up the programming language?
4. **Plans and Structures.** How are the words of the programming language used to express ideas, plans, and algorithms?
5. **Pragmatics and Tools.** How is a program entered into the computer, executed, and how does it interact with its environment?

Different approaches to computer science education address one or more of these five areas of difficulties. For instance, Turtle graphics presents the student with a very simple notional machine, and the programming language "Logo" reduces the syntactic overhead to a minimum. More recent approaches such as "Scratch" try to eliminate the difficulty of syntax altogether through a block-based programming language.

Where programming language and machine are given, new environments are created, primarily addressing the area of pragmatics and tools in du Boulay's list. These environments work with an existing programming language, and strive to make it more accessible to the novice user. "BlueJ" and "Greenfoot" (see, e. g., [30] and [31]), for example, offer an easier access to Java and object oriented programming. An advantage of such educational environments is that they are not limited to assisting the student in learning the correct syntax and semantics, but also make the underlying notional machine more accessible. This can be done, for instance, by the means of a debugger or improved compiler messages.

TigerJython is such an education environment for Python programming. Its main characteristics are more detailed error messages in case of syntax errors, and a debugger that exposes the internal state of the machine while a program is being executed. Moreover, it also includes a set of educational libraries, for instance the above mentioned Turtle graphics.

Design goals. The development environment TigerJython has been required to, and fulfills the following list of design goals. Apart from a focus on didactical considerations, it has always been important to reduce technical hurdles to a minimum.

- The environment must have an intuitive and simple interface. In particular, after starting, students must immediately be presented with an editor window to start typing their program code. Executing a program must be just as intuitive and simple.

  In contrast, Python's standard editor "IDLE" requires the user to explicitly create a new file first, and save its contents, in order to execute the program. Most professional environments typically start by creating a project.

- The environment must be available on a wide range of systems. In particular, it must run without installation, so that it can be used on systems where the user has restricted rights, and cannot install new software (as, for instance, on a computer system provided by the school or institution). It must also run on different operating systems and platforms.

- All necessary libraries, in particular the needed graphical libraries, must be included. A typical user should not be required to download or install any additional libraries. At the same time, using external and additional libraries must still be supported.

- A debugger allows the student to execute the program stepwise line by line, and inspect the machine's state at any time. The machine's state needs to be visualized in an accurate

and accessible way.

An important aspect of the debugger is that it follows the idea of the "notional machine". In order to run a Python program, the internal registers of the CPU, for instance, are not important because they are not accessible from, or part of the Python language.

- In case of (syntax) errors, the system reports intelligible error messages. On one hand, this is to say that the error messages are to report the actual problem as precisely as possible. On the other hand, TigerJython offers its error messages in English and German – the latter being the native language of our student population.

2.2 Building Upon Classroom Experience

2.2.1 Input and Output

Python's interactive console. Python provides an interactive console/terminal (also known as "REPL"), which allows the user to enter Python code, and thereby evaluate expressions, and execute statements. The environment included with standard Python is called "IDLE" and shown in Fig. 2.1. For the subsequent discussion, it is important to note, that IDLE integrates user

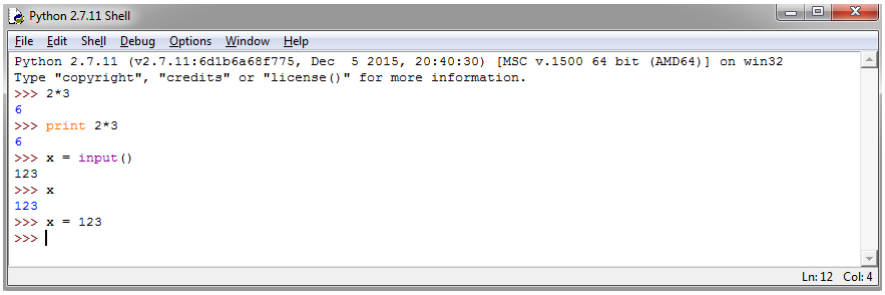


Figure 2.1: A screenshot of the interactive console "IDLE", that is included with standard Python.

input and output into the interactive console. As can be seen in the screenshot in Fig. 2.1, the console does not show any difference between evaluating the expression `2*3` and executing the statement `print 2*3`. Moreover, even the `input()`-function takes its input from the interactive console. In this setting, the advantages of using `input()` instead of assigning a value directly to a variable remains somewhat obscure.

Print versus return. Our students exhibited difficulties understanding the concept of the `return` statement. On first sight, indeed, the `return`-statement shares a main characteristic with `print`: to "return" the result of a computation. Consider, for instance, the following two examples:

```
def sqr(x):
    return x*x
print sqr(12)
```

```
def sqr(x):
    print x*x
print sqr(12)
```

The output of both programs is almost identical. In the case of `print` on the right, Python prints an additional `None`, which is simply ignored by the students.

When either of the `sqr(x)`-functions above are invoked from the interactive console/terminal, the output is exactly the same, and no difference is visible at all. It is understandable, then, that students have difficulties understanding why they should use `return` instead of `print` – especially given that in a program (as contrasted to the terminal), `return` does not produce any visible output.

In order to help students discern between `print` and `return`, we introduced a dedicated output window for all "printed" output. While values directly evaluated appear in the console, and can be used for subsequent computations, printed values do not appear inside the console, and have no effect on further comutations. This can be seen in Fig. 2.2.

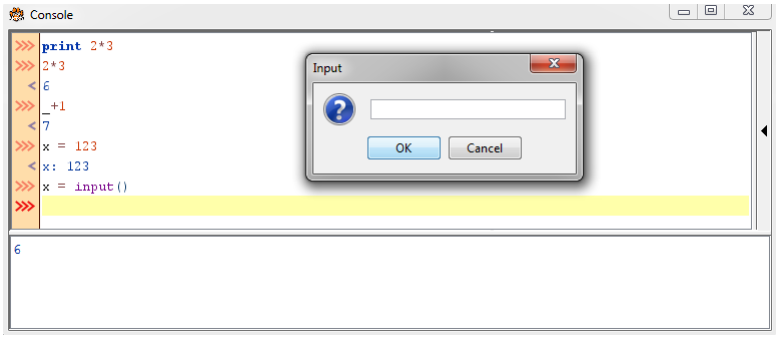


Figure 2.2: The interactive console as it is included in TigerJython. Note how the output of the first statement is to be found in the dedicated output window at the bottom, and how input is done through a small dialog window.

Visible and invisible results. Another aspect of this is that Python does not automatically print evaluated expressions when running a program – there an explicit `print` is required. Consider, for instance, the expression statement `2*3`. Inside the console, it is evaluated, and the result `6` is shown in the console. Inside a program, however, the result of the expression is not printed, and hence does not appear anywhere to be seen.

Input. When using the `input()`-function in standard Python, students frequently asked the teacher for assistance because their programs seemingly stopped working. Indeed, apart from printing a possible prompt, Python does not give any clue that user input is expected (see Fig. 2.3).

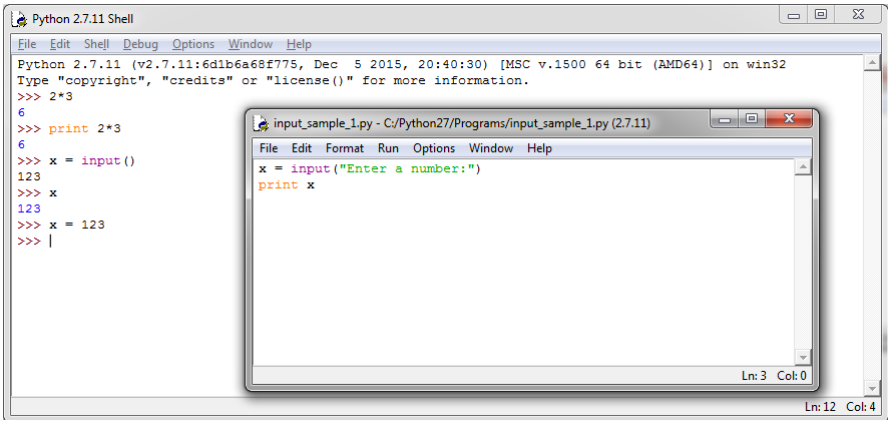


Figure 2.3: There is hardly any indication that the Python program shown in the small window is currently running, awaiting user input (due to an error in Python, not even the prompt is showing correctly).

To make requested user input clearly visible to the user, TigerJython uses small dialog boxes for all user input (see Fig. 2.2). This has the additional benefit that students clearly see an effect when using an `input()`-function, and understand the difference to putting constant values into the program code.

In Python 2, the `input()`-function interpretes the given input as a Python expression, and attempts to evaluate it. As long as the user enters numbers, this comes in very handy, as the returned values of `input()` are, indeed, the numeric values of the entered numbers. In case of string values, however, the user is confronted with a "NameError" as seen in Fig. 2.4.

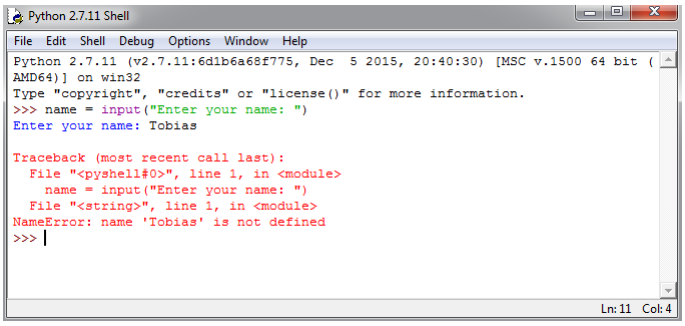


Figure 2.4: In Python 2, the input function evaluates its input as a Python expression. In case of an entered "name", it is interpreted as the name of variable, leading to an error.

As an alternative, Python offers the function `raw_input()`, which always returns the entered text as a string value. In this case, the user can safely enter names, say, but numbers are then also returned as strings, and not as numeric values.

TigerJython uses a different concept for the `input()`-function. All numbers are returned as numeric values (either integer or floating point values). Everything else is returned as a literal string value (see also Section 3.5).

2.2.2 Location of Error Messages

Quite some work has been done considering the question of what information an error message should contain, and how that information should be presented (see, e. g., Becker [5], Denny et al. [11], Marceau et al. [39], Nienaltowski et al. [43]). However, the placement and presentation of the error message itself is of some importance as well.

Our experience with IDLE. IDLE shows *syntax errors* in the editor window by marking the offending line in red, and by displaying a small dialog window with the error message, that must be closed before editing the Python program. *Runtime errors* are shown in the interactive console, usually as "Traceback" exposing the call stack (see Fig. 2.5). In classroom we observed

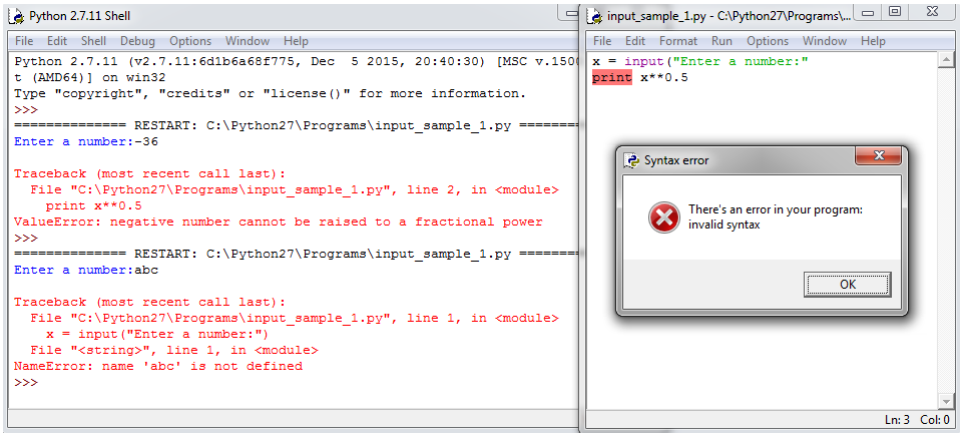


Figure 2.5: Python's environment "IDLE" displays runtime error messages inside the console (on the left), and compile time errors (i. e. syntax errors) in a small dialog window (on the right).

three points about IDLE's way of displaying errors.

- Students tended to overlook the runtime errors shown in the console. In particular, it seems that these messages were not directly regarded as helpful information about the error in the program, but rather as some technical details with no further significance. In case of a problem, students usually concentrated on the editor window with the program code.
- In case of the dialog window, students closed it before reading its contents, and then, again, concentrated on the program code in the editor.

- Many students reported that they would not read the error messages because the messages were in English (our students were native German speakers). In fact, most error messages are even given in a technical English, and many students might not understand what “no viable alternative” or “syntax error” actually means.

Designing better ways to display error messages. TigerJython’s way of displaying errors has been based on the experience with IDLE. Moreover, this way has changed as we gained further experience with TigerJython itself. Our goal has been to design error messages that are seen, read, and understood by students without a lengthy introduction to error messages and their meaning.

We found that error messages have to be placed directly into the editor, so as not to be overlooked by the students. Dialog windows forcing the students to confirm the error message did not work as expected. The main problem arose when students needed further assistance from a teacher. In these cases, they asked for help after having closed the window, and thus eliminating all information about the error. Hence, the error messages need to remain accessible until at least the next run of the program. This led to a design where error messages are both shown directly inside the editor, as well as recorded in a dedicated window at the bottom (see Fig. 2.6).

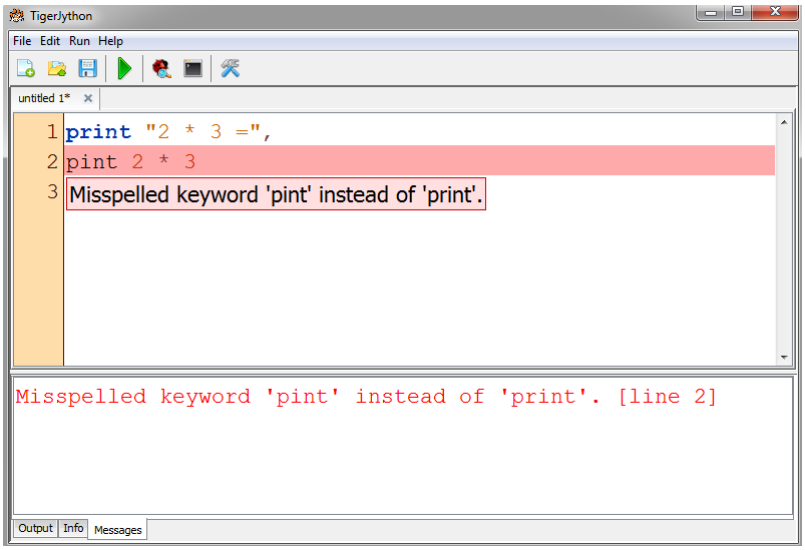


Figure 2.6: Errors in TigerJython are displayed directly inside the editor, as well as in an error window at the bottom.

In order to make error messages more intelligible to the students, we translated most messages to German. In this process, we also enhanced the error messages by adding additional explanations and information. This enhancement, however, proved to be detrimental in most

cases. Students were unlikely to read longer messages. This is also confirmed by a study of Nienaltowski et al. [43]: "Giving a lot of information in an error message does not necessarily help students get the correct answer." Accordingly, we subsequently simplified the error messages: TigerJython now displays rather concise messages.

We conducted a survey among the students, assessing the benefits of TigerJython and its error messages. The survey is discussed in Section 8.4. Most notably, students reported that the German translations of the error messages were, indeed, helpful.

2.3 Jython

Jython [28] is an implementation of the Python language that runs on the Java Virtual Machine. At the time of writing the current version supports Python 2.7. Jython's main strength is its interface between Python and Java. This allows, in particular, for Python programs to fully interact with, and leverage the power of, Java libraries.

From a technical point of view, using Jython allows us to create a single executable Java file, containing all necessary libraries and dependencies. Accordingly, TigerJython does not require any external software apart from a Java Runtime Environment, and runs on various platforms without installation.

Benefits of using Jython. Jython is a well tested and complete implementation of Python. Using Jython as the underlying system means that TigerJython is not a "toy system" reduced to a few educational examples, but offers (almost) the entire range of Python's possibilities.

Due to Jython's interface to Java, we have been able to reuse libraries originally written for Java. With very little porting effort and overhead, such libraries could be made available to the students. The Turtle graphics library used in TigerJython, for instance, was originally written for programming in Java.

Interaction of Python and Java. There are some minor differences between standard Python and Jython, due to the fact that Jython interacts with the Java Virtual Machine. These differences include the import or modules, packages and classes, function overloading, or using unicode characters in string values. We found, that some of these differences gave rise to problems, as documented below.

In fact, the correct handling of unicode characters in strings has proven to be particularly problematic. A string returned from Java to Python can cause an exception because of non-ASCII-characters contained within.

A Python program executed in Jython has full access to Java. This has two profound implications regarding the notional machine. First, the machine's state is no longer exclusively determined by values (directly) accessible to Python. Java methods can manipulate the state

of the underlying Java Virtual Machine. Second, a Python program might spawn new threads, which continue to run even after the Python script itself has finished. This is particularly visible, for instance, when a Python program opens a graphical window. Jython might then report that the Python program has terminated, yet the window remains fully active, and can even call functions from within the Python script.

2.3.1 Controlling Program Execution in Jython

Stopping programs. Jython supports a "trace callback" function: we can register a function to be invoked before each line of Python code is executed. This trace callback provides the basis for stopping a program, as well as for executing a program line by line.

Inside the graphical user interface, the user is given the possibility to stop and abort the execution of a Python program. Jython, however, does not directly provide a way to stop a Python program. However, by throwing an exception, the trace callback can cause a program to terminate immediately – unless, of course, the currently running part of the Python program is enclosed in a `try-except`-block. Due to the design of this mechanism, a program is actually stopped at the beginning of the next line to be executed. If the current statement, however, invokes a Java method, Jython has to wait until that method has terminated before the program can be aborted. This applies, in particular, to `sleep()`- and `input()`-functions, which both delay the continuation of program execution.

Unfortunately, the technique for stopping the program does not work in case of other threads accessing the Python script. Consider the following Python program to be run in TigerJython.

```
from gturtle import *

@onMouseClicked
def mouseClicked(x, y):
    while True:
        pass

makeTurtle()
```

The program registers a callback function `mouseClicked`, and then opens up a window. After the last statement `makeTurtle()`, Jython reports that the program has terminated, even though the window is still open and active. When the user clicks into the window, the callback function from this Script is still executed. This is even done in Java's "Event Dispatch Thread", responsible for all user interface related actions. In other words, this program effectively blocks the entire user interface.

The trace callback is actually invoked only as long as the Python program is run by the "main executing thread". Even if the call to `mouseClicked` was not performed by Java's "Event Dispatch Thread", Jython is not aware that the function's code is executed at all. Hence, the development environment has no way to react.

Debugging and stepwise execution. The trace callback described above is also used in order to update the debugger, and to execute the program step by step. Delaying or pausing a program is achieved by a loop inside the trace callback. After the user has asked the program to pause, the trace callback delays further execution of the program until the user has given the command to continue.

Moving forward and back. In his "Online Python Tutor" [19], Philip Guo offers the possibility to not only execute and trace a program step by step, but also to move forward and back in the program execution. This is achieved by prior execution of the entire program, where, after each step, the machine's state is recorded. What the user then perceives as execution of the program, is actually just moving through the list of previously recorded machine states. Guo's Online Python Tutor is restricted to a maximum of 300 steps.

In case of TigerJython, such an implementation proved to be virtually impossible. Recoding the machine's state so as to allow moving a step back imposes two essential restrictions onto the programs at hand. First, it must be possible to completely capture the machine's state. Second, the program cannot interact with the user. As already mentioned above, a Python program in Jython might interact with the entire Java environment. This includes the possibility to create and open graphical windows, or to interact with yet unknown libraries. Capturing the entire state of the machine is impossible under these circumstances. In addition, even simple functions such as `input()` become problematic, again strongly restricting the possibilities of the environment.

Because of these severe limitations, TigerJython does not offer an option to move backwards in program execution.

2.3.2 Changes to Jython

Out of educational reasons, we have made some changes to Jython. From a technical point of view, the changes to the programming language (most notably, the introduction of a **`repeat`**-loop) are achieved by a preprocessor. Additionaly, we changed input- and output-behaviour, and added some functions to be available directly without any imports. More explicitely:

- TigerJython has a **`repeat`**-loop to repeat a code sequence for a given number of iterations.
- While Python uses the `^`-operator for logical exclusive or (xor), the operator has often become to denote power, i. e. `a^b` stands for a^b. The software package "SageMath" [55] is based on Python, but redefined the `^`-operator to denote the power-operator, and `^^` for exclusive or. TigerJython supports this semantical change of `^` as an option as well.
- The `input()`-function always prompts the user for input using a small dialog window. At the same time, the input is not evaluated as Python expression. However, numbers are

returned as numeric values. In addition, the functions `inputInt()` and `inputFloat()` are extensions of `input()` requesting an integer, or any number, respectively.

- Additional standard functions have been defined. These include, most notably, `msgDlg()` to display a text message in a small Dialog window, `makeColor()` to create colors for Turtle or any other Java-based graphics, and `playTone()` to play sounds and musical tunes.
- Since the standard Turtle-module of Python is not available in Jython, we replaced it by an extended Turtle-module, based on a Java-library by A. Plüss [46].

Further details about the **`repeat`**-loop, as well as input are given in Section 3.5.

2.4 Debugger

In an educational environment, a debugger's primary purpose is not necessarily as a means to find and correct errors. Rather, a debugger can be used in order to visualize a machine's state and the flow of code and data during program execution. The debugger of TigerJython has been built with this purpose in mind.

2.4.1 Variables and Types in Python

This section provides only a limited overview. A more detailed discussion of how Python handles variables and types can be found in Section 3.3.

Variables, types and immutability. Python is dynamically and strongly typed. That is to say, Python's variables are untyped, while the values carry a fixed type. In other words: the type of a variable can change any time, but the type of a value never changes. As seen from the Java-side of Jython, we can consider each variable as having a generic type `PyObject`. The variable's values are then specific subclasses of `PyObject`, such as, e. g., `PyString` or `PyInteger`.

Many values in Python are actually *immutable*, and thus cannot change. Let's consider, e. g., a variable x with the integer value of 3. This is to say, that x is a reference to an "integer object" carrying the value 3 (in Jython, the object is of type `PyInteger`). The Python code `x += 1` might then be expected to increment the object's value by 1. However, this is not the case. Rather, Python creates a new integer object with the value 4 and has x reference that new object.

The difference between mutable and immutable types becomes apparent in the case of parameters. The two following programs illustrate this point with an immutable integer type on the left, and a mutable list type on the right. In the case of the integer type, the program will print "3", since the statement `p += 1` causes the parameter p to reference a *new* object. In the case of the list type, however, the program will print "`[3, 1]`". In this case, it is not the variable (reference) p that has changed, but rather the underlying list object itself.

```
def foo(p):
    p += 1    # p refers to new value
x = 3
foo(x)
print x
```

```
def foo(p):
    p.append(1)   # p's value changed
x = [3]
foo(x)
print x
```

In Python, each variable is a reference to an object. In case of assignments, it is the reference that is copied, not the object itself. However, when talking about immutable types, there is no discernable difference between using a "primitive" type (i. e. the variable carrying the value itself), and referencing the object. For educational purposes, we can simplify the model of references and objects, and instead think of `x += 1` as directly increasing the value of the variable.

Scope and frames. Each module and function is executed in the context of a "*frame*". Most importantly, a frame holds a dictionary (table), assigning to each name or variable its proper value object. This dictionary or table is called the "*locals*" dictionary, referring to the fact that it holds "local variables".

With each invocation of a function, a new frame is created. The new frame has a reference to the current frame, and starts out with a locals dictionary containing the function's parameters together with their values (the arguments). An assignment to a variable creates, or replaces an entry in the current frame's locals dictionary.

Differences in Jython. Since Jython runs on the Java Virtual Machine, and strives for compatibility with Java, there are some differences between standard Python and Jython.

Jython usually performs automatic type casts between most Python and Java types. When calling a Java method, for instance, requiring two floating point values as parameters, and returning a string value, Jython converts both the parameters from Python to Java types, as well as the result returned by the method from Java to Python.

Even though the use of Java types is typically transparent to the Python programmer in Jython, Jython internally distinguishes between Python and Java types. When designing a debugger, we must take care to correctly take into account the additional wealth of types, stemming from the interaction between Java and Python. For instance, methods and functions can behave quite differently: not only does Java clearly distinguish between static and virtual methods (but has no proper functions), it also supports function overloading. That is, in Java, a function can have different implementations, discerned by the parameters used to invoke them. In Python, these methods must be represented by a single function, that will invoke the correct method when called.

2.4.2 Displaying Frames and Variables

The design of the debugger's display of variables and names includes a trade off. Adhering to Python's concept of variables as references would impose complex graphs of objects and their relationships. At the same time, the debugger must capture essential features of Python's notional machine to be useful in an educational setting.

Figure 2.7 shows TigerJython's debugger during the execution of a (trivial) program. Only the list `my_list` is a mutable object, and thus displayed as a true object, with the variable `my_list` being a reference to the object. The tuple `t`, even though also a compound object, is shown as a "value" rather than an object, emphasizing that its value cannot be changed.

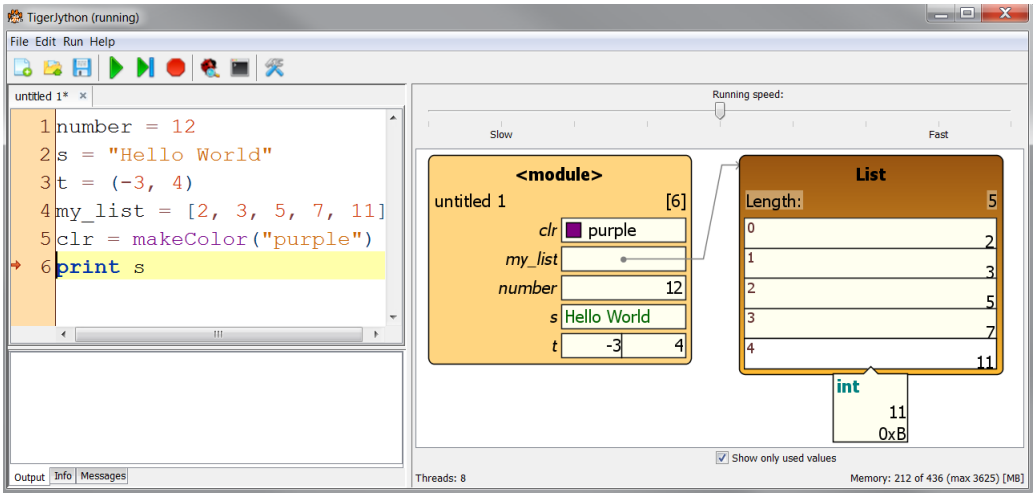


Figure 2.7: The debugger on the right displaying different data types. Note, in particular, the difference between the immutable "`Tuple`"-type, and the mutable "`List`"-type. In addition, when the mouse hovers over one of the values, additional information is shown (on the right bottom).

The debugger has two columns. The left column shows the frames, the right column additional objects. The current and active frame is alwas placed on top, as seen in Figure 2.8.

Importing modules might result in a huge list of available names, rendering it very difficult to find those variables which actually play a role in the program. TigerJython therefore filters the names, and shows only those names which are actually used in the program. This can be seen in Figure 2.8, where only the three functions `forward()`, `left()` and `makeTurtle()` are shown in the module's frame (even though, in actuality, all turtle functions have been imported).

Extended information. For a few selected data types, the debugger shows additional information whenever the mouse hovers on top of the respective value. The selection of such extended information is primarily based on didactical considerations.

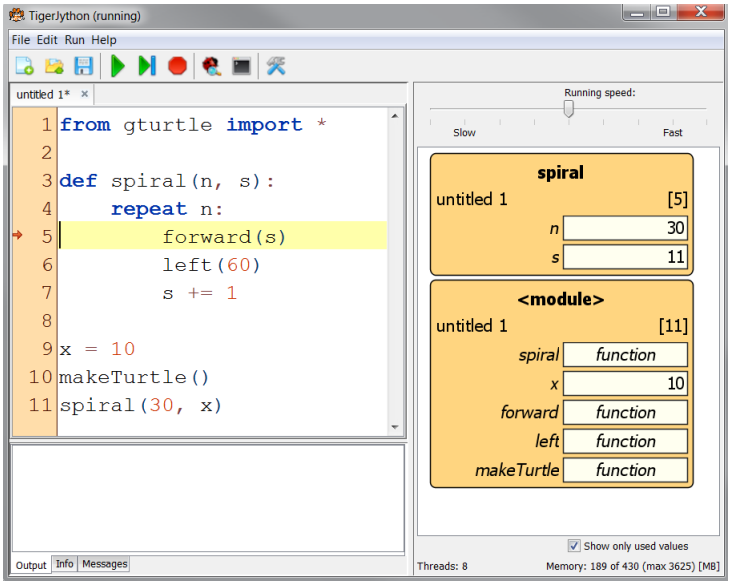


Figure 2.8: The debugger (on the right) during program execution. As program flow is currently inside the "spiral"-function, there are two frames: the upper one representing the function, the lower one being the global frame for the entire module. Note that only functions and values are displayed, which are actually used in the Python program.

For example, even though Python does not have a data type for characters, the debugger recognizes strings of length 1, and not only displays the actual character, but also the corresponding character (ASCII) code, both decimal and hexadecimal. This is intended to help with programs that rely on each character having a certain integer code, such as the Caesar cipher.

Another example of extended information is shown in Figure 2.9. The debugger does not only display the name and the hexadecimal value of a color value, but also its decomposition into the three basic colors *red*, *green*, and *blue*. This is intended to help students experiment with, and understand, how colors are represented by a computer.

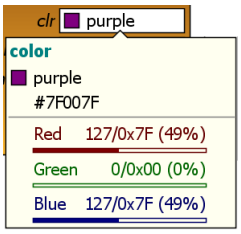


Figure 2.9: For color values, the debugger shows the composition of red, green and blue.

Chapter 3

The Python Programming Language

3.1 Introduction

This dissertation is based on the programming language *Python*. As the reader might not be familiar with Python, this section provides an overview and introduction as needed for the rest of the thesis. This overview cannot be exhaustive, of course. The interested reader will find more comprehensive discussions in the official documentation [48, 51].

We assume that the reader has a general understanding of imperative programming, although not necessarily in Python. Hence, the reader should be familiar with concepts such as variables, data types, functions, statements, loops, conditionals, etc.

There are currently two versions of Python in use: Python 2 and Python 3. The two versions are not fully compatible, but differ, among other things, in syntactical details and the included standard libraries. This thesis uses Python 2 as its basis, more precisely in the distribution *Jython*, which runs Python 2.7 on the Java platform [28].

Changes to Python. The Python environment *TigerJython*, that has been developed along this thesis, introduces some changes to standard Python. They are documented in section 3.5.

Most notable, we have added a `repeat`-statement to provide the novice programmer with a loop that does not require variables. The other changes primarily concern built-in functions such as `input`.

3.1.1 A Short Summary of Python's Basic Features

- Statements are separated by line breaks. However, inside parentheses and brackets, line breaks are ignored. Semicolons can be used to separate simple statements, with the effect that semicolons at the end of a line are usually ignored.

- Code blocks (i. e., the bodies of functions, loops, etc.) are marked by indentation. An indented code block is always preceded by a colon at the end of a compound-statement's header (e. g., `if x > 0:`).
- Functions are defined with the keyword `def` and always return a value. If no value is specified by using a `return`-statement, the function returns `None` (which roughly corresponds to "void", "null", etc. in other languages).

  In order to call/invoke a function, parentheses are always needed, even when no arguments are given. A function's name without parentheses refers to the function as an object.
- Python is dynamically and strongly typed: variables carry no type information, while all values have a fixed type. In addition, Python is object-oriented: all values are objects, and variables are of generic type "object".

  Accordingly, variables require no declaration. A new variable is created when a value is assigned to it.
- Strings can be enclosed in single or double quotes: `'abc'` and `"abc"` are equivalent. There is no "char"-type for strings of length 1. Strings that begin and end with three consecutive quotation marks can span multiple lines.
- Integers can have arbitrary size, hence Python's `int` corresponds to "BigInteger" in other languages and systems.
- Python makes heavy use of tuples, e. g., in order to swap the values of two variables: `(a, b) = (b, a)`. Tuples are also used to return multiple values from a function, e. g., `return (1, 2, 'abc')`.
- Comments start with the hash-character `#` and run until the end of the line.

3.1.2 Python's Terminology

Lines Python's syntax allows for multiple **physical lines** to be connected to a single **logical line**. For instance, line break tokens are ignored within parentheses and brackets, so that the definition of a list, for instance, can run over several physical lines and still only count as one logical line in Python.

Statements Python distinguishes between **simple** and **compound statements**. Compound statements include other statements as part of their body. Examples are loops, functions, etc.

Simple statements are a sequence of *small* statements, which are separated by semicolons. An example for a simple statement is `x = 3; print x`, comprising an assignment and a `print`-statement. Simple statements with more than one small statement are rather uncommon in Python and the semicolon is rarely used.

Suite A block of code, as part of *compound statement*, is called a **suite**. Hence, the `while`-loop would be defined as:

⟨*while_stmt*⟩ ::= 'while' ⟨*expression*⟩ ':' ⟨*suite*⟩

The suite is either a simple statement (possibly comprising several small statements) or an indented block of code, following the `while`-clause on the next line.

Names Identifiers are often called **names** in Python.

Expressions While **expression** usually refers to an *arithmetic expression*, a **test** can either be an arithmetic expression, a comparison, or even a conditional expression.

3.2 Examples of Python Programs

This section contains five examples to illustrate the use of Python.

Drawing a square with the turtle. This first example shows how to draw a simple square using the turtle. A program like this can usually be written by a high school novice programmer within the first two hours of the course. Both the `gturtle`-module as well as the `repeat`-loop are additions of "TigerJython" not found in standard Python.

```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        left(90)

makeTurtle()
square()
```

Using tuples. This tiny example shows how tuples can be used to return multiple values from a function. Also note how the entire `if`-statement in line 2 is on one line. The suite – in this case just a `return`-statement – may be just a simple statement following the colon instead of an indented block.

```
def mini_sort(a, b):
    if a > b: return b, a
    return a, b

minValue, maxValue = mini_sort(12, 34)
```

Random walk with the turtle. Using the pseudo-random generator, we can have the turtle perform a random walk. In our case, the turtle is restricted to a circle of 200 pixels around the screen's center.

Note the different ways to import a module. While each of the four modules (including the `math`-module) is loaded only once, the imports differ in how the names are imported into the current namespace, as can be seen by the use of `math.sqrt`, `time.sleep` and `random.randint`, respectively.

```
from gturtle import *
from random import randint
import time

def distance(x, y):
    import math
    return math.sqrt(x**2 + y**2)

makeTurtle()
hideTurtle()
repeat 100:
    left( randint(0, 360) )
    forward(20)
    dot(8)
    if distance(getX(), getY()) > 200:
        dot(10)
        heading(towards(0, 0))
        forward(20)
    time.sleep(0.1)
```

A "Brainfuck"-interpreter. Brainfuck is a minimalistic programming language, comprising only eight symbols (`<>+-[].,`) [57]. Similar to a Turing machine, it operates on an (infinitely large) tape of cells. The following Python program is an interpreter for the Brainfuck programming language, however, without the input-function. It interprets the Brainfuck program given in the string below and prints the "Hello World"-message.

In order to modify the global variable `index` from within a function, the variable has to be declared as global. This is not necessary for the list `cells` since the program does not change the variable `cells` but rather the referenced list itself (cf. Section 3.3).

Another peculiarity of Python is that, for any sequence `s`, `s[-1]` returns the last element of the sequence.

```
cells = [0] * 30000  # Our "tape"
index = 0            # The index to the current cell
stack = []           # An empty list

def move_index(c):
    global index
    index += 1 if c == '>' else -1
```

```
def change_cell(c):
    while index >= len(cells):
        cells.append(0)
    cells[index] += value

def interpret(text):
    i = 0
    while i < len(text):
        ch = text[i]
        i += 1
        if ch in ['<', '>']:
            move_index(ch)
        elif ch in ['+', '-']:
            change_cell(int(ch + "1"))
        elif ch == '.':
            print chr(cells[index]),
        elif ch == '[':
            stack.append(i)
        elif ch == ']':
            if cells[index] != 0:
                i = stack[-1]
            else:
                stack.pop()

input_text = """
   >+++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.+++++++..+++.>>>++++++++
   [<++++>-]<.>>>++++++++++[<+++++++++>-]<---.<<<<.+++.------.--------.>>+.
"""
interpret(input_text)
```

Gaussian elimination. This example follows as closely as possible a Pascal program given by N. Wirth in [59]. It uses Gaussian elimination to solve the system:

$$\begin{bmatrix} 1 & 2 & 5 \\ 3 & 1 & 4 \\ -2 & 5 & 9 \end{bmatrix} \cdot \vec{x} = \begin{pmatrix} 4 \\ 11 \\ -7 \end{pmatrix}$$

Note that a `for`-loop in Python always iterates over the elements of a sequence (e. g., a list, a string or a tuple). The `range`-function returns a list of successive integer values. Hence, `range(4)` is equivalent to the list `[0, 1, 2, 3]` and `range(3, 7)` to the list `[3, 4, 5, 6]`.

```
n = 3
A = [[1, 2, 5], [3, 1, 4], [-2, 5, 9]]
B = [4, 11, -7]

for k in range(n):
    p = 1.0 / A[k][k]
    for j in range(k, n):
        A[k][j] = p * A[k][j]
    B[k] = p * B[k]
    for i in range(k+1, n):
```

```
        for j in range(k+1, n):
            A[i][j] = A[i][j] - A[i][k] * A[k][j]
        B[i] = B[i] - A[i][k] * B[k]

k = n-1
X = [0 for i in range(n)]
while k >= 0:
    t = B[k]
    for j in range(k+1, n):
        t = t - A[k][j] * X[j]
    X[k] = t
    k -= 1

for i in range(len(X)):
    X[i] = round(X[i], 6)
print X
```

3.3 Variables and the Type System

In any imperative programming language, at runtime, each variable holds (or is a reference to) a value. Depending on the language, we can then attribute a certain type to the variable, the value or both. In Python, each value has a fixed type, whereas the variables may reference any value, and hence type. This makes Python a *dynamically* and *strongly* typed language.

Statically typed languages fix the type of variables and parameters. This allows to easily infer the type of any variable already while the program is being written and compiled, and before it is actually executed. Since Python lacks statical typing, a compiler cannot, in general, infer and reason about the types of the variables involved.

In constrast to weakly typed languages, a value cannot change its type in a strongly typed language, and the only automatic type cast is from “integer” to “floating point” numbers. For instance, in the weakly typed JavaScript language, the expression `"1"+ 2` evaluates to the string `"12"` whereas Python throws an exception because of incompatible types.

Dictionaries. One of the most basic data structures in Python are *dictionaries* (also known as “associative array” or “hash-table”), as will become apparent in the course of this discussion. Basically, Python uses dictionaries to manage variables and fields. This allows, e. g., for dynamic creation of new variables anywhere in the program.

Objects. From an object-oriented perspective, Python’s values are all objects descending from a common base class `PyObject`. As the variables all carry the same type `PyObject`, we can omit type declaration in the program.

All operations are internally translated to method calls on the objects. For instance, the expression `x + 1` internally translates to `x.__add__(consts.int(1))`. Type checking is therefore

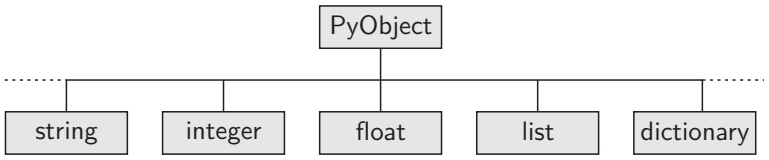


Figure 3.1: Python's types are in fact classes with a flat hierarchy as shown here. However, every value in Python is also an object of a common ancestor type "PyObject".

done by the objects themselves and not by the interpreter.

In addition to standard methods such as the `__add__`-method mentioned above, each object holds a dictionary with additional fields or methods. A method call or field access from within Python is performed by looking up the respective name in this dictionary. If a method or field cannot be found, the interpreter reports an "object has no field"-error.

Frames and the lifetime of variables. Whenever a code block with its own scope is entered (such as a program/module or function), the interpreter creates a *frame*-object to hold all variables during the code's execution. Each frame has a dictionary with its "local" variables, as well as references to a calling frame (if any) and the currently executed line.

The lifetime of any variable is certainly limited to the lifetime of the containing frame. However, variables are dynamically created whenever a value is assigned to it. For instance, the statement `x = 2` might create a new variable `x` in the current frame, and assign the value 2 to it. There is also the possibility of deleting a variable so that its lifetime ends before the end of the scope.

Finally, Python's `exec`-statement even supports the creation (or removal) of variables and functions with names determined at runtime. For instance, we could set a variable x to the value 123.45:

```
exec "x = 123.45"
```

It is immediately clear that the contents of such a string can be determined at runtime. This method is used, for instance, by Python's standard `turtle`-module, where almost all functions available through the module are created dynamically.

The bottom line is that a Python compiler has not only difficulties in inferring a variable's type, but it might even be unable to reliably determine all variables and functions that are defined at any given point in the program.

Mutable and immutable values. Each value is an object, and each variable is a reference to an object. This implies that the copy assignment `y = x` does not actually copy the object itself but just creates a new reference to it. For immutable objects, including numbers and tuples, there is no discernable difference to actually copying the value. Some objects, such as lists, however, can

change their value(s).

For example, the following program will print `[1, 2, 3]`.

```
x = [1, 2]
y = x
y.append(3)
print x
```

Strings and integers. Note that Python does not have a dedicated "char"-type, but uses strings of length 1 instead. For integers, Python provides two types: `int` for 32-bit integers and `long` for integers of arbitrary size. For most programmers, however, the distinction between these two types is completely transparent and newer versions of Python have dropped the distinction altogether.

3.4 Python's Grammar

This section provides a brief overview of Python's grammar as far as it is relevant for the remainder of this thesis. We naturally leave out various details, which can be found in Python's official documentation [48, 51] (the full grammar specification can also be found in Appendix 9.2).

The primary purpose of this section is to provide the basis, upon which we can discuss the parser in Chapter 7. We will therefore include some additional remarks that are relevant in the context of parsing and error detection.

Python 2 versus Python 3. Python 3 brought a series of changes to the syntax and semantics of the languages. For instance, `print` is no longer a reserved keyword, but has become a mere built-in function. On the other hand, `True` and `False` have become proper keywords. While a discussion of the differences and merits is beyond the scope of this thesis, it is important to note that different versions of Python exist.

In order to ease transition, Python allows to change `print` from a keyword to a function already in Python 2. This is done via a special import-statement. In other words, the grammar can actually be changed by the program code.

Lines and indentation. In Python, each line contains basically one statement. However, inside brackets and parentheses ⟨*NEWLINE*⟩-tokens are ignored. Hence, a *logical line* can be distributed across several *physical lines*. The following is an example of a single logical line spanning two physical lines, due to the fact that the ⟨*NEWLINE*⟩-token is inside brackets, and hence ignored.

```
primes = [2, 3, 5, 7,
          11, 13, 17, 19]
```

Each logical line has an *indentation* property, which is the number of white space characters at the beginning of the line. The indentation is used to form code blocks (suites) and express

control flow. In the following example, the "`forward`" is repeated 4 times, whereas the "`left`" is not part of the loop's body.

```
repeat 4:
    forward(100)
left(90)
```

Python's grammar specifies that the lexer uses a stack to keep track of the lines' indentation. The stack starts with a zero on top, that is never removed. For each line L, its indentation I_L is compared to the indentation value I_S on top of the stack. If I_L is larger than I_S, the lexer pushes I_L onto the stack and produces a ⟨*INDENT*⟩-token to mark the beginning of a suite. If I_L is smaller than I_S, the lexer pops values off the stack and produces ⟨*DEDENT*⟩-tokens until I_S and I_L are equal. Accordingly, Python's official grammar specifications uses ⟨*INDENT*⟩- and ⟨*DEDENT*⟩-tokens, respectively.

3.4.1 Expressions

Expressions and tests. The basic syntax of *expressions* follows standard rules as found in many modern programming languages. In addition to *expressions*, Python also introduces *tests*. A *test* might be one of the following (this is also expressed in Fig. 3.2):

- an *expression*, e. g., `foo(1, 2) + 3 * items[4]`,
- a *comparison*, e. g., `0 < foo(1, 2) <= 9`,
- a Boolean expression, e. g., `x > 4 and y < x`,
- a conditional expression, e. g., `1 if x >= 0 else -1`,
- an anonymous function, e. g.: `lambda x: x**2`.

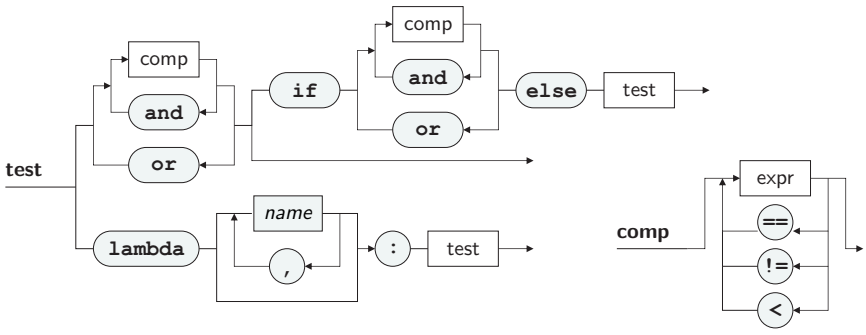


Figure 3.2: A *test* might just be a simple *expression*, or a sequence of expressions and comparison operators. It also includes the possibility of conditional expressions or anonymous functions.

Sequential types and their literals. Lists and dictionaries are among the most important types in Python. Both types can, of course, be defined using a literal syntax such as `[1, 2, 3]`, or `{'a': 1, 'b': 2}`, respectively. There is, however, the additional possibility of using *list comprehension*.

Using list comprehension, we can write the list of all square numbers from 1^2 to 9^2, and the list of all odd square numbers, respectively, as:

```
digits = [1, 2, 3, 4, 5, 6, 7, 8, 9]
squares = [x**2 for x in digits]
odd_squares = [x for x in squares if x % 2 != 0]
```

A list comprehension is always an expression, followed by a `for`-structure and possibly `if`-expressions. Dictionary comprehensions differ only in the expression at the beginning (and of course the curly braces instead of the square brackets), e. g.:

```
ascii = { chr(x) : x for x in range(65, 91) }
```

Finally, note that list comprehensions can also be used as arguments in a function call.

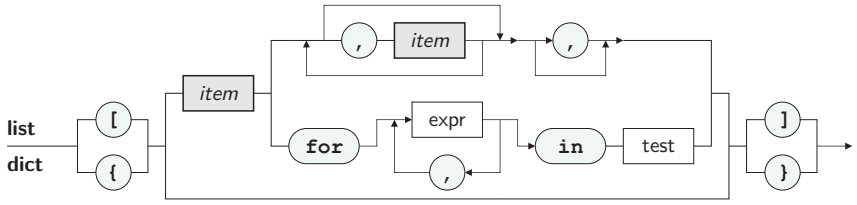


Figure 3.3: Besides the usual sequence of items a *list* (or *dictionary*) could also contain a *list comprehension*. For a list the *item*-node here is a "`test`", while for dictionaries it is a "`test ':' test`".

Parsing and error recognition. Three keywords that are otherwise thought of in the context of statements can also occur as part of expression: "`if`", "`else`" and "`for`". Accordingly, they cannot necessarily be used as synchronizing tokens, and it is more difficult to diagnose the correct error in cases such as `x = z if (y)`. This might be either an incomplete `if`-expression or an extra space in the function name "`zif`".

The actual grammar for list comprehension is rather delicate with some obscure special cases. Python's official grammar mentions that both the two following expressions must be legal [48]:

```
[ x for x in lambda: True, lambda: False if x() ]
lambda x: 5 if x else 2
```

In case of an error in a slightly more complex list comprehension, correct identification of the problem might just be virtually impossible.

3.4.2 Statements

Compound and simple statements. Python distinguishes between *compound statements*, which themselves can contain other statements, and *simple statements*. Typical examples of compound statements include function definitions, if-statements and loops, each having a body of other statements (the suite). Simple statements, in turn, can contain several *small statements*, separated by semicolon.

The body of compound statements is called *suite*. A suite is either a simple statement directly following the colon, or an indented block of statements below the compound statement's header. That is:

⟨*suite*⟩ ::= ⟨*simple_stmt*⟩ | ⟨*NEWLINE*⟩ ⟨*INDENT*⟩ ⟨*stmt*⟩+ ⟨*DEDENT*⟩

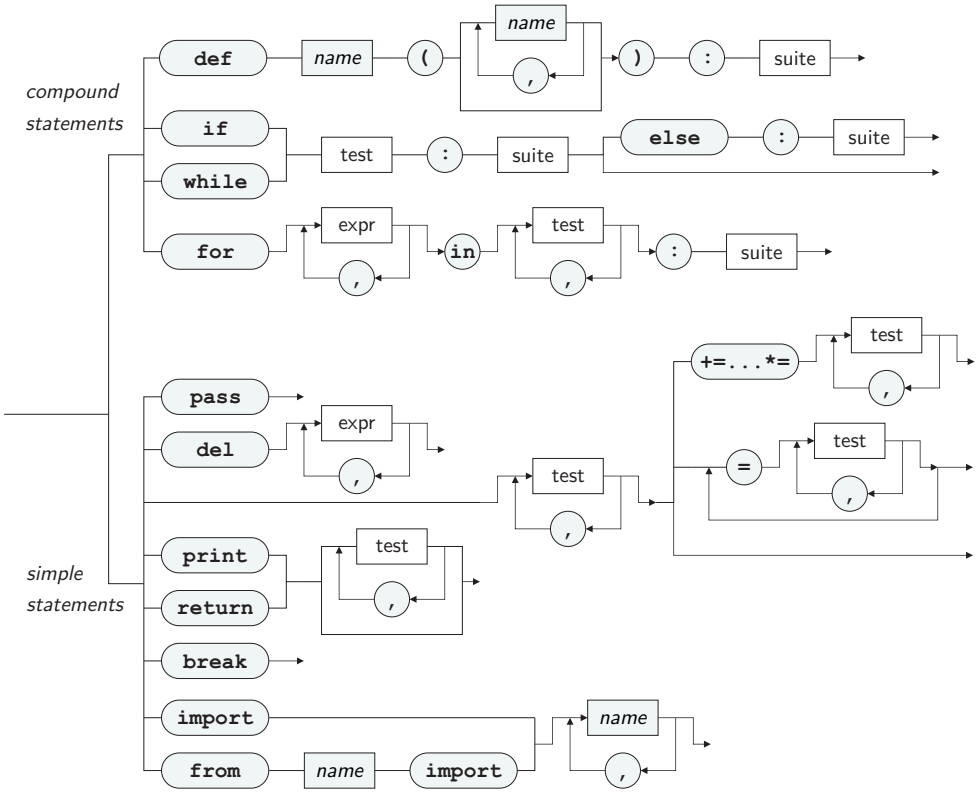


Figure 3.4: A simplified version of statements in Python, with *compound statement* at the top and *simple* or *small statements* below. For simplicity, this overview leaves out some details (such as default values for parameters in functions) and a few statement-types (such as classes and try-catch-structures).

Expression statements. Every test and expression is also a valid statement. The expression can be followed by assignment operators such as "=" for direct assignment or "+=" for augmented assignment. This means that both "`x = 1`" (an assignment) and "`x == 1`" (a simple test) are valid statements in Python, even though the second one has absolutely no side effect (it still is compiled and executed, though).

Discussion. In principle, almost all statement types in Python can be identified by their initial keyword. The exceptions only include assignments and expression statements, most notably function calls. However, these exceptions typically make up a large portion of actual programs. Moreover, since we cannot assume the Python program to be syntactically correct, we must find alternative ways to identify the various statements.

For our purposes it is important to note that in particular compound statements cannot only be identified by their initial keyword but also by their structure (apart from a few ambiguous cases such as "`ef foo():`", which could be both a function definition or an if-statement). On the other hand, keywords such as **`if`**, **`for`** and **`in`** serve different purposes, depending on context. When trying to correct a program we must therefore be careful not to confuse, e. g., if-statements with if-expressions.

3.5 Changes to the Python Programming Language

As part of the educational programming environment *TigerJython*, we have made some changes to the Python languages as used in the environment. Most notably, we introduced a new looping structure for educational purposes, and changed the behaviour of the built-in `input`-function.

Division. The division of two integer values can, in generall, yield two distinct results: the result may be either another integer value or a floating point value. For instance, $14/5$ may yield 2 (with remainder 4), or 2.8, respectively. Python 2 discerns between the two possible result types, based on the types of the two operands. If both operands are integer values, the division is interpreted as an integer division yielding an integer result. Otherwise, the division yields a floating point number. In other words. $14/5$ yields 2, but $14.0/5.0$ yields 2.8.

A variable's datatype, however, is solely based on the value. This makes it very difficult to predict for any two numeric values the true outcome of the division operation. Python 3 rectified this situation by introduction of two different division operators. In Python 3 the `/`-operator always yields the floating point result, whereas the `//`-operator is used to obtain the result of an integer-division.

For the educational TigerJython-environment, we decided to use the two division-operators as in Python 3. This is not entirely compatible with traditional Python 2 implementations, but gives more consistent and predicitable results.

Input. Python 2 has two separate input-functions. `raw_input()` returns the user's input as a string value. `input()`, on the other hand, evaluates the input as a Python expression and returns the result thereof. When a numeric value is entered, `raw_input()` returns the string representation of the number, while `input()` returns the number itself. For proper names, however, `input()` is problematic as it tries to interpret the entered name as a variable in Python's current context.

For educational purposes, it is desirable to have an input-function that returns numeric values as either integer or floating point values, and everything else a string value. For the TigerJython-environment, we therefore changed the `input()`-function, so as to return either the numeric value as integer or floating point, respectively, and everything else as a literal string.

A simple loop. Python's grammar includes two looping structures: **`for`**- and **`while`**-loops, respectively. In order to repeat a code sequence a given number of times, both structures may be used. In this case, however, both structures use a variable.

From an educational point of view, it might be desirable to introduce variables at a much later point than loops (in particular when taking into account the difficulties of the variable concept, cf. Chapter 5). The programming language "Logo", for instance, provides a simple loop structure that works entirely without variables. This allows to introduce the concepts of loops and variables seperately [25].

We have therefore introduced a new loop structure into our Python environment: a **`repeat`**-loop takes a number and then repeats its body for the given number of times. The grammar had to be adapted as follows:

⟨*compound_stmt*⟩ ::= ⟨*old_compound_stmt*⟩ | ⟨*repeat_stmt*⟩

⟨*repeat_stmt*⟩ ::= '`repeat`' [⟨*expression*⟩] '`:`' ⟨*suite*⟩

Internally, a **`repeat`**-loop is translated to a **`for`**-loop.

Finally, the **`repeat`**-loop can be used without a number to iterate its body indefinitely, i. e., to create a potentially infinite loop. Semantically, such a **`repeat`** is equivalent to **`while`** `True:`.

Chapter 4

The Models of Mathematics and Programming

4.1 Introduction

Mathematics and imperative programming use different underlying models of computation. In a nutshell, mathematics both transforms expressions (terms) and uses substitution on a syntactic level to simplify expressions and thereby arrive at "evaluated" values. Programming, in contrast, performs computations on concrete values and objects.

In order to make the differences between the two fields more accessible, we discuss the concepts of *function* and of *variable* in mathematics and programming. Understanding the differences between mathematics and imperative programming will later allow us to state to which extent students apply a mathematical model to programming.

In practice, the vast majority of students will not have a completely developed mental model of mathematics as presented here. The overall concepts and ideas, however, still apply and help us understand how students could look at programming.

$x = x + 1$. The formula $x = x + 1$ is most famous for having completely different interpretations in mathematics and programming. Mathematics interprets the formula as an equation equivalent to $0 = 1$. There is no term we can substitute for x that would make the formula a true statement. In imperative programming, on the other hand, the formula is a computational statement, meaning that the current value of the variable x shall be increased by 1.

At first glance, it looks as if the problem of $x = x + 1$ could be solved by introducing a new asymmetric assignment operator (e. g., "<–" or Pascal's ":=" [40, 58]). There are, however, underlying misconceptions that cannot be tackled by a mere change of syntax. Students might,

e. g., believe that a variable can hold an entire expression or even equation. Bayman and Mayer found in a study about novice programmers' misconception in Basic programming that a third of the students understood the statement `LET A = B + 1` as writing the expression `B + 1` to memory space `A` [4].

If we consider the statement `x = x + 1` as storing the expression $x + 1$ in variable x, we still get an unsolvable problem. For example, would the following program actually print anything?

```
x = 3
x = x + 1
print x
```

When the "value" of x is the expression $x + 1$ we can perform a substitution in the third line, still ending up with the expression $x + 1$ containing the variable x. This variable would then have to be substituted by $x + 1$ again, leading to an infinite expansion:

```
print x => print x+1 => print x+1+1 => print x+1+1+1 => ...
```

Hence, the statement $x = x + 1$ is not only problematic for students who believe that it is an equation to be solved, but also for those who have a mathematical model of the machine where variables are substituted by expressions.

Finally, even if the assignment is fully understood as storing a numeric value in a variable the statement "`x = x + 1`" has a sequential oddity. Whereas virtually all expressions are evaluated from left to right, the assignment statement must first evaluate the expression on the right. Even though we understand the increasing of a variable's value by 1 as an atomic statement, it must take two steps in time in the present form (unless a compiler transforms it to an atomic instruction).

In conclusion, the rather common and essential operation of increasing the value of a variable is far from trivial. Several papers propose to explicitly teach the role of variables and common patterns such as this statement to increase a variables' value (e. g., Samurçay [52], see also Section 5.2). The approach of this thesis, however, is to rather try and correct the underlying model programming students use to explain what happens when a computer executes the program.

Related work. Various articles already mention the phenomenon of students inappropriately applying mathematical concepts in programming, particularly in the case of variables.

Samurçay states that the concept of variable in programming is a new concept for the students and is in contrast to the mathematical, but insufficient concept they already have [52].

Putnam et al. [47] describe in their article about misconceptions of programmers how a student thinks that `LET X = X + 1` was not possible because the variable X had previously been set to zero, leading to the statement `LET 0 = 0 + 1`. They then remark: "*This student's misconception appears to be the result of inappropriately bringing knowledge from another domain – algebra – to programming.*" [47].

However, even though the connection between mathematics and programming is frequently

mentioned, to our knowledge there is no explicit study about students applying a mathematical model to imperative programming.

4.1.1 Organization

The chapter's core is to be found in Sections 4.2 and 4.3, respectively. In these two sections, we discuss the concepts of variables both in mathematics and programming. This will be the basis for further discussion in the remainder of the thesis.

Section 4.4 includes an additional discussion about the concepts of functions in mathematics and programming. Even though we will not make explicit use of the function concepts, we chose to include this brief discussion here to further highlight and elaborate the differences between the two models that underlie mathematics and programming, respectively.

4.2 Variables in Mathematics

As based on logic and set theory, mathematics is a system of transformations on the syntactical level. Simplifications such as $a^3 \cdot a^4 = a^7$ are not based on what the symbol a is or what value it might represent. It is a truth directly obtained from the syntactical definition $a^3 = a \cdot a \cdot a$. In fact, it is a strength of mathematical reasoning that such simplifications are applicable to any object for which we have the concept of an associative multiplication.

Basic definitions. For a complete definition of the entire framework see, e. g., Halbeisen [21]. Since we do not require the full language of logic, we restrict ourselves to a small subset which we deem relevant for the current discussion.

The formal language of mathematics includes (but is not limited to) the following symbols:

- *Variable symbols*, e. g., $x, y, z, \ldots$
- *Constant symbols*, e. g., $0, 1, \ldots$
- *Function symbols*, e. g., $f, +, \circ, \ldots$

Each function symbol has an *arity* n which is the number of its arguments. Binary operators such as $+$, for instance, are 2-ary function symbols. In this case, instead of "function", we will also use "operator" as a synonym.

A *term* is then a word in the formal language according to the following rules: any variable or constant symbol is a term, and, for any n-ary function f, the expression $f(t_1, t_2, \ldots, t_n)$ with terms t_k is a term.

Given any two terms t_1 and t_2, we can build an equation $t_1 = t_2$, meaning that any occurrence of the term t_1 can be replaced by the term t_2 and vice versa (in practice, we might need parentheses to avoid ambiguity).

According to the above definitions of the formal language, a variable is just a symbol in the language's alphabet. However, this is missing one important aspect of variables as captured by the following rule: *Let x be a variable symbol in a term φ. We can then substitute all occurrences of x in φ by a term t, written $\varphi(x/t)$. The result $\varphi(x/t)$ is also a term.* (Note that this is a simplification in which we leave out the notion of formulae, as well as of free and bound variables).

In a nutshell, a *variable* is a symbol that can be substituted by a term. The substitution is performed on a purely syntactical level.

An example from group theory. In group theory, we always have an associative operation $\circ$ and a (left) neutral element e, such that, for any element g in the group, we have $e \circ g = g$. This means that, whenever we have a term g, we can replace it by $e \circ g$ and whenever we have $e \circ g$, we can replace it by g. The same applies to the inverse element g^{-1}, and the equivalence $g^{-1} \circ g = e$.

Based on these basic definitions, the additional equivalence $g \circ g^{-1} = e$ (which would have to be proven beforehand), and the idea of substitution, we can prove that e is also a right neutral element:

$$g = e \circ g = g \circ g^{-1} \circ g = g \circ e$$

Note how the entire proof uses only substitution of symbols. At no point did we require the elements g, g^{-1}, or the operation $\circ$ to be anything beyond mere symbols.

In many applications of group theory, we will replace $\circ$ by another operator symbol such as $+$ or $\cdot$, and then use either 0 or 1 as the neutral element. While this looks as if we were to use concrete values in such cases, we have still not given an interpretation for the constant symbols 0, 1, or the operator symbols. 0 might denote the first element of the natural numbers $\mathbb{N}$ or a vector $(0, 0, 0)$, say. The constant symbol 1 is often used not only to mean the natural number, but also for the unity matrix, say.

Solving equations. Solving an equation for a variable x means that we want to find all constant symbols, for which we can substitute x such that the equation becomes a true statement. Depending on the domain, there does not always exist such a constant. The equation $x^2 + 1 = 0$, for instance, has no solutions in real numbers, $x = x + 1$ has no solution at all.

Yet, even in the case of unsolvable equations, we can still perform typical transformations such as subtracting x on both sides in the case of $x = x + 1$, yielding $0 = 1$. We can even try and transform $x^2 + 1 = 0$ into (the not entirely equivalent equation) $x = \sqrt{-1}$. Even though there is no value for $\sqrt{-1}$ in the real numbers, it is a legal term in mathematics.

Our point is: variables are mere symbols adhering to some simple rules. If variables had to be placeholders for existing objects, we could neither write down an equation such as $x = x + 1$ nor transform it.

This is exactly where imperative programming differs: during execution of a program, each variable must represent an existing value or object.

4.3 Variables in Programming

In the context of programming, the value of a variable is a property of the computing machine's current state. Within each state, a variable's value is a fixed constant, but as the states change a variable can take on different values.

The concepts presented here are in part also based on the formal definitions of Abstract State Machines. The formal definition of Abstract State Machines is based on formal logic and includes a thorough definition of what a variable is (see, e. g., Börger and Stärk [7] or Gurevich [20]).

Computational machines. There are various models for computational machines, including the Turing machine, the simpler finite state machine (FSM), or the random-access machine (for a more detailed overview, see, e. g., Savage [53]). Even though the Turing machine provides the common basis for questions about computability, real computing devices can be modelled as the simpler FSMs, since they lack an infinite storage capacity. Savage remarks: "*Although there are languages that cannot be accepted by any machine with a finite number os states, it is important to note that all realistic computational problems are finite in nature and can be solved by FSMs.*" [53]

For our present discussion about concepts in imperative programming, taking the FSMs as basis of the computational machine will suffice. A reader more familiar with Turing machines might just as well think of the computational machine as a Turing machine. We will, however, speak of the machine's *states*, which would correspond to the more inclusive concept of *configurations* in the case of Turing machines.

The essential property of our computational machine is that it can be seen as having a set of states Q, along with well defined transitions between states. By programming the machine, we specify rules for these transitions.

Variables. In the context of the states of a computational machine, we can understand what a variable in imperative programing is. The most important aspect is that a variable is a constant symbol as defined on the current state. Changing a variable's value always includes a change to another state. Hence, for each state q, a variable gives us a uniquely determined value attached to that state q.

More formally, a *variable* is a function $f : Q \to T$ from the set of states to a non-empty set T. For each state q, it returns a uniquely determined value in T.

The set T is usually modelled as the variable's data type. Hence, for a variable of type integer, T might be the set of all integer values between, say, $-32\,536$ and $32\,535$.

As an example, let us consider a simple computer with two 8-bit registers. We identify the machine's states with 16-bit numbers as concatenations of the two registers, resulting in states such as q_{0105}, q_{0A1F}, etc. (where we have used hexadecimal numbers to shorten the representation of the 16 bits). The two registers could be represented as functions $r_A : Q \to T$ and $r_B : Q \to T$, respectively. In this case, T is the set of all integers from 0 to 255. For the state q_{3C12}, we then obviously get $r_A(q_{3C12}) = 3C$ and $r_B(q_{3C12}) = 12$, respectively.

Finally, note that, in contrast to mathematical logic, a variable $f\colon Q \to T$ always returns a constant symbol from the set T and not a general term. Hence, for any given state, a variable is not a symbol subject to substitution, but a value with fixed meaning, instead.

Changing context. Once we have established that the values of variables represent (part of) the state of the computation machine, we can easily interpret assignments to variables as a particular type of transition. Conceptually, assigning a value to variable like $x = 2$ could be seen as "find and proceed to the state where the variable x takes on the value 2."

It is essential to understand that all statements in the program operate on a given state. In this context the initially considered example $x = x + 1$ is not a (mathematical) statement about x or any of its properties, but rather an instruction for how to change the program's state.

The change of state during a program's execution renders it a necessity to keep track of the state if we want to (mentally) trace a program. In contrast to mathematics, a statement like $y = 2x$ does not hold universally in the entire program, but has to be applied on a specific state and must be seen in that context. Or, considering that variables are functions (i. e., $x = x(q)$), we can see the statement $y = 2x$ as "*continue the program in state q with $y(q) = 2x(q)$*". For any subsequent state q', however, the equation $y(q') = 2x(q')$ does not necessarily have to hold.

4.4 Functions

The defining property of a function in mathematics is that, for each set of inputs, the returned object or value is uniquely determined. In imperative programming, this requirement of uniquely determined results is dropped. Indeed, in a program we can easily have a function with no arguments at all that returns a different value each time it is evaluated – think, for instance, of a function that returns the current time in milliseconds.

Mathematical functions as static objects. In the language of set theory, a function $f : M \to N$ with sets M and N can be identified with a subset $G \subset M \times N$. G then has to fulfill the property that, for any two elements $(x_1, y_1), (x_2, y_2) \in G$, we have $x_1 = x_2 \Rightarrow y_1 = y_2$. The subset G is sometimes called the *graph* of the function (see, e. g., Amann and Escher [2]).

In some cases, we find a formula to express the relationship between input $m \in M$ and output $n \in N$ of a function $f : M \to N$. We should be aware, though, that it is not the act of

calculation that assigns the output n to the input m. The tuple (m, n) is an element of the graph G in an absolute sense and "timeless manner", and the calculation of n for a given m is merely the search for that particular tuple. In mathematics, it does not matter how we obtain the value n.

Mathematical logic introduces yet another aspect of functions. In this context, a function f with parameters $t_1, t_2, \ldots, t_n$ is a term whenever we replace each of the parameters t_k by a term. The value of a function is here thought of as being constructed from its parameters on a mere syntactical level. There is no actual computation involved (see, e. g. Halbeisen [21]).

Computations. In contrast to mathematics, the notion of computation is essential to a function in programming. The value of a function is computed from the values assigned to the parameters and the current state of the machine. Evaluation of a function does not take place on the syntactical level as in mathematics and logic. More to the point, a function such as $f(x) = x^2 + 1$ returns the concrete, computed value 290 for the input 17, whereas in mathematics we could say that $f(17)$ primarily equals $17^2 + 1$.

In mathematics, we consider the terms $T_1(x) = x^2 + 7x + 3$ and $T_2(x) = (x + 7)x + 3$ as equivalent. For any real number $x \in \mathbb{R}$, both T_1 and T_2 yield the same value. From the computational point of view, however, there is a difference. The first term T_1 has four operations including two multiplications. The second term T_2 comprises three operations with one multiplication.

A function in programming may depend not only on the arguments but also on the current state of the machine. This is evidenced by functions returning a pseudo-random number, input values from a peripheral device or some properties of the machine's state such as the current time. Due to this implicit dependency on the machine's state and possibly external processes, a function's value is not in the same way uniquely determined as in mathematics.

Of course, as long as we consider computational machines as closed deterministic systems (i. e., without input from an outside world), we can again regard functions as uniquely determined with the machine's current state as an implicit parameter.

4.5 Conclusion

Core concepts such as variables and functions clearly differ in their meaning when seen from a mathematical or programming perspective, respectively. While mathematics focuses on syntactical manipulations, programming emphasizes computations and hence the changing of a computational machine's state.

The mathematical model of substitution breaks down in programming because each occurrence of a variable has to be seen in its proper state and context. In short, a variable in programming is in fact a function of the state and assignments can be seen as transition rules.

Chapter 5

An Investigation of the Concept of Variables in the Context of Programming Education

5.1 Introduction

Computer science and mathematics share a common history and have a close relationship. This is also reflected in a common terminology. Especially in the context of imperative programming, however, some common terms shared among both disciplines refer to different concepts. Among these terms are *function* and *variable* as discussed in Chapter 4.

The foremost example to illustrate the difference is the famous statement "$x = x + 1$". In imperative programming this statement is an assignment, increasing the value of the variable x by 1 (assuming that x refers to a positive integer in the first place). In mathematics it would be interpreted as an equation with no solution. There is no object we could put in for x that would render this statement true.

The similarities and differences between mathematics and programming become important in high school programming education. We found indication that some students apply mathematical concepts to programming, resulting in misconceptions about how programming works.

Based on the concept of *cognitive conflict* and by teaching the students how to manually trace the values of variables in the program, we succeeded in improving their understanding of variables and the computational machine.

5.1.1 Theses

In the course of this chapter, we are going to provide support for two of our theses:

Thesis 1. Some common misconceptions and errors made by novice programming students can be explained as the students applying a mathematical model of syntactical substitution to program execution.

Thesis 2. It is possible to directly improve the students' understanding and cognitive concept of variables and the computational model through explicit teaching.

5.1.2 Organization

This chapter is organized as follows.

Section 5.2 summarizes related work about programming students' misconceptions. Of special importance are Sections 5.3 and 5.4 as they present and discuss our own findings about these misconceptions.

In Section 5.5, we will present a possible solution to improve students' understanding and to foster them in developing a correct mental model.

5.2 Related Work about Students' Misconceptions

The novice programmer has been extensively studied and the literature spans several decades of research in different areas. For on overview of recent research, see, for instance, the articles by Robins et al. [50], and Pears et al. [45]. Still one of the most frequently cited sources on this topic is Soloway and Spohrer's collection of papers [54].

We limit our discussion of related work to variables and assignments, and include only studies that are of direct relevance for our own work.

5.2.1 Studies about Misconceptions

Syntactic problems. The foremost syntactic problem is certainly the perceived symmetry of the equal sign, letting student believe that both $x = 2$ and $2 = x$ were valid assignments. This is confirmed by several studies. Another problem pertaining the syntax is the belief that the name of a variable has direct influence on its contents, e. g., a variable called "max" would automatically hold the maximum value of a list.

Du Boulay remarks that the asymmetry of the assignment operator = confuses some learners and that they have difficulties seeing why the statement "`LET A = 2`" is legal while "`LET 2 = A`"

is not [13]. McIver and Conway [40], for instance, suggest that a language for novice programmers should use an operator such as "$\leftarrow$" instead of the equal sign "$=$".

The perceived symmetry of the equal sign is also mentioned by Putnam et al. [47]. They note that students occasionally interpreted "`LET A = B`" as assigning the value of A to B (from left to right). However, students had no problems with statements such as "`LET A = B + C`" where the direction is more clear and less ambiguous.

The `READ`-statement found in the programming language Basic seems to have been the source of various difficulties and misconceptions. Students would, for instance, believe that the variables "smallest" and "first" in the following program would hold the values -3, and 99, respectively [47].

```
READ SMALLEST
READ FIRST
DATA 99, 2, -3, 6, 29
```

Modern programming languages usually have no equivalent to such a "`READ`"-statement. The notion that the variable's name determines its content might therefore be partly outdated.

This dissertation is based on Python as the programming language. Changing the syntax to address possible difficulties is therefore beyond its scope. It could be an interesting study, however, to see whether changing the assignment operator has any effects on the students' misconceptions and would improve their performance.

The values of variables. For a novice programmer it might not be clear what kind of information a variable actually stores. Some students believe that a variable keeps a history of all its values while other students assume that a variable can store entire expressions.

Putnam et al. [47] state that the most significant misconception concerning variables is that a variable could hold more than one value. This misconception mostly surfaced in the context of "`READ`" and "`PRINT`"-statements. Printing the value of a variable, for instance, would print the entire history of values ever stored in that particular variable.

Of particular interest in the context of our study is the notion that a variable might hold an unevaluated expression. Du Boulay indicates that a student might understand the assignment "`LET A = 7 + 4`" to store the unevaluated expression instead of the value 11. He proposes to tackle this misunderstanding "*by stressing the idea that a variable can hold only one number*" [13]. However, our students are trained in mathematics classes to consider "$7+4$" as "a number". Writing, e. g., $7 + 4 \in \mathbb{N}$ is common practice in mathematics. Just stressing that a variable can only hold one number might therefore not be enough to correct any misconceptions.

The nature of assignments. It is frequently noted that students seem to think of assignments as equations. More interesting in our context, though, are questions related to the exact nature of assignments when understood as assignments. While some students believe `x = y` links the

two variables together (reflecting changes in each other), others assume that the source variable y would be empty after the assignment.

It might be somewhat problematic that, in some programming languages or under certain circumstances, `x = y` could mean that the two variables are linked together. If both x and y are pointers or references, for instance, any change to x would also affect y. In Python, this is true when y is a mutable object such as a list.

Bayman and Mayer [4] asked 30 college students to describe the working of the statement "`LET A = B + 1`". While 30 % of these students gave a correct answer, 43 % said that the equation $A = B + 1$ would be written to memory. 33 % of the students believed that the expression $B + 1$ would be stored as the value of the variable A (note that some of the students obviously had more than one concept in their answers). In other words, a third of the students understood the concept of an assignment (in contrast to an equation) and still believed that the unevaluated expression would be stored as a variable's "content".

In his discussion about the statement "`LET A = B`", du Boulay states that some novices see the statement as linking the two variables together "*so that whatever happens to 'A' in future also happens to 'B'*" [13]. He also notes how it could be seen as removing the contents of B and placing them in A, and concludes: "*One of the problems for the learner is distinguishing an operation which implies copying, and so independence, from one of sharing and so dependence*" [13].

The roles of variables. The basic idea of studying the roles of variables is based on the observation that variables are often part of more or less fixed structures. For instance, a variable might be used as a counter variable in a loop. Teaching such roles of variables to students could improve the students' understanding of variables.

In his paper on the concept of variables in programming, Samurçay notices that "*The mathematical model of variable and the equality relation constitutes for a novice an initial but insufficient model for operating on the programming variables*" [52].

Sumurçay's approach to improve the understanding of variables is based on the idea that variables in programming often have specific roles, based on plans or structures. He discerns three different uses of variables in a loop: initialization, update and test. In addition, he proposes four different uses of the assignment sign:

1. Assignment of a constant value, e. g., `a := 3`,
2. Attribution of a calculated value, e. g., `a := 3 * k`,
3. Duplication, e. g., `a := k`,
4. Accumulation, e. g., `a := a + 1` or `a := a * k`.

According to Samurçay the "*concepts of variables and assignment take their full programming meaning in the last case*" [52], hence in the case of accumulation.

While we certainly agree that programming students often start out with a mathematical model of variables, we do believe that even "simple" statements such as duplication in the above list can be a cause of confusion and misconceptions.

The idea of the different roles of variables has been picked up by several studies [9, 32, 44]. The general consensus is that the explicit teaching of the roles of variables increases the students' understanding and performance.

The idea of different roles of variables can also be found in the literature about mathematics education (e. g., Heck [22]). Our focus, however, is more on the underlying concepts and mental models held by the students. We therefore did not pursue this further as part of this thesis.

5.2.2 Recent Studies on Difficulties

More recent studies focus more on the difficulties of individual concepts than on the students' misconceptions about these concepts. Studies like the one of Lathinen et al. present surveys on how difficult students (or tutors) rate concepts such as variables or loops [15, 33, 41]. While these studies certainly provide valuable insight, in the context of difficulties and misconceptions we cannot be sure how accurate the students' sel-assessments are. In fact, students who are not aware of their own misconceptions might rate a topic or concept as easy and still not be able to apply it correctly (cf., e. g., Muller et al. [42]).

Probably more accurate in identifying the difficulties of programmers are studies about the students' actual performance. Lister et al., for instance, analyzed students' reading and tracing skills of programs [35]. Yet, their conclusion that they "*see few comprehension errors due to misconceptions*" [35] might not apply to our own case. The programs used in the study did not include situations where a misapplied mathematical model would really become apparent. In particular, there were no assignments of more complex expressions to variables.

5.3 Students' Misconceptions about Variable Assignment and Evaluation

5.3.1 Methodology

The author of this thesis has been teaching an introduction to programming for over four years at a Swiss high school. The class is mandatory during 10th grade for all students who choose physics and applied mathematics as their elective. The entire course comprises two semesters with two hours each week. All classes take place in a computer lab and the emphasis is on hands-on exercises. Due to space restrictions in the computer lab, the number of students is typically

between 10 and 16 students. Larger classes are split among different teachers.

During each semester, the students take three graded tests with a total of six tests per year. The tests have to be written without access to a computer. In order to assess the overall understanding of the students, the author has collected selected answers from the tests. They form the basis for the following presentation.

From the collected answers, we selected four problems to be included in our investigation. In order to be considered for inclusion in the study, the problems' solution had to include non-trivial use of variables. In particular, we were looking for dependencies between different variables. The four problems then included are the following.

[P1] The students had to write a program that computes the solutions to the quadratic equation $ax^2 + bx + c = 0$. The students were explicitly required to handle special cases.

[P2] The students had to write a program that draws the graph of the mathematical function $f(x) = x(x-5)(x+5)/25$ in the range between -40 and 40.

[P3] The students were given an incorrect solution to [P2] and were asked to trace the program and identify the error.

[P4] The students were asked to trace a given program including variables and functions, and explain the reasoning behind their respective answers.

5.3.2 Summary

From the problems included in the tests, we chose four problems to present here. For each problem, we state the number of students who completed the test, together with a brief description of the question. We then reprint the original question (please note, however, that the original questions were in German) and give a report of the students' answers as they are of interest for our study.

The results from the four problems can be summarized as follows.

[P1] **Solving the quadratic equation.** Four of twenty students computed a value before checking whether the value can be computed. This might be an indication of a model of lazy evaluation where a variable's value is not actually computed until it is used.

[P2] **Drawing the graph of a function.** Six of twenty students did not recompute the value of a dependent variable (another ten students did not provide any usable answer and are therefore not included in the statistics). This is consistent with a model where variable assignments establish a relationship between different quantities.

[P3] **Analyzing a program that should draw the graph of a function.** Six of ten students did not correctly trace the given program. Their answers are also consistent with a model where variable assignments establish a relationship between different quantities.

[P4] **Tracing a program involving variables and functions.** Four of ten students argued that a variable's value would be computed at the time the variable is actually used. Their reasoning is consistent with both a model of lazy evaluation and one where variable assignments establish a relationship between different quantities.

The numbers of students exhibiting a model of lazy evaluation or of assignments establishing relationships between variables are summarized in the following table.

[P1] Quadratic equation	4	20	20 %
[P2] Graph of a function (1)	6	20	30 %
[P3] Graph of a function (2)	6	10	60 %
[P4] Tracing values	4	10	40 %

5.3.3 Quadratic Equations [P1]

Twenty students were asked to write a program to solve quadratic equations, handling special cases. For our study, the interesting question is whether the students check if the solutions can be computed before actually computing them.

Problem: Write a program to solve the quadratic equation $ax^2 + bx + c = 0$. Make sure you handle special cases such as when no solutions exist. Hint: use the formula you know from mathematics class:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Students' answers: In principle, there are two special cases requiring attention when using the above formula. First, the value of a could be zero, leading to a division by zero. Second, the discriminant $D = b^2 - 4ac$ might be negative, leading to non-real results.

In order to solve the problem students were expected to test for the case of a negative discriminant before computing the square root of it. While most students did indeed check for the special case, not all of them did this checking *before* computing the square root. The following program shows part of a student's answer. Note that the solutions x_1 and x_2 are computed before the check for a negative discriminant in line 3.

```
1 x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
2 x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
3 if b**2 - 4*a*c > 0:
4     print "The solutions are", x1, x2
5 elif b**2 - 4*a*c == 0:
6     print "The solution is", x1
7 else:
8     print "There are no solutions"
```

Some students also checked if the square root of the discriminant was zero:

```
x = (-b + sqrt(b**2 - 4*a*c))/(2*a)
if sqrt(b**2 - 4*a*c) > 0:
    print "The solution is", x
```

For our study we only looked at whether a student did some "fragile" computing before checking for computability. The correctness of the test itself was not important, so that both of the programs presented above are considered as "computing before test".

Frequency: The problem was included in the tests of two distinct classes with ten students each. Four of the ten students in the first class made the particular mistake of computing the solution first and checking for computability afterwards. In the second class, after a more thorough discussion of the problem, no student made this mistake.

5.3.4 The Graph of a Function (1) [P2]

Twenty students were asked to write a program that draws the graph of a given (mathematical) function. The interesting question for our study is whether the students recompute the dependent variable after changes were made to the independent variable.

Problem: Write a program to draw the graph of the function $f(x)$ given below in the range between -40 and 40.

$$f(x) = \frac{x(x-5)(x+5)}{25}$$

Students' answers: While there was some minor disagreement among the students' answers whether the loop should iterate 80 or rather 81 times, the overall answers were very similar. Some students, however, did not recompute the value of the dependent variable y (representing $f(x)$) inside the loop as would be required for a correct program. The following program shows a student's answer where y is only computed once.

```
x = -40
y = x *(x-5)*(x+5) / 25
setpos(x, y)
for i in range(80):
    x += 1
    lineto(x, y)
```

Frequency: From a total of 30 students ten did not provide an answer at all or calculated the necessary values by hand to insert them as constants into the program. Since we have no information to assess their mode of thinking, we did not include them in our statistics. Among those 20 students providing a usable answer, six did not recompute the value of y inside the loop.

5.3.5 The Graph of a Function (2) [P3]

Eleven students were given a program, asked to determine the picture drawn by the program, and to find a logical error in the program. This is basically the inverse problem to the first "graph"-problem: students are given an incorrect solution, and are asked to figure out why it does not work.

Problem: (a) What picture does the program given below actually draw? Sketch the resulting picture. (b) The program contains a logical error. Describe briefly what is wrong and how the program can be corrected.

```
1 x = -20
2 y = 1/4 * x**2 - 9
3 setpos(x, y)
4 repeat 41:
5     lineto(x, y)
6     x += 1
```

Correct solution: (a) The program draws a horizontal line. (b) In order to draw the intended parabola, the value of y would have to be recomputed inside the loop.

Students' answers: Of the eleven students completing the test, one did not provide an answer to this question. Six of the remaining students said that the program would draw a parabola and four students said the program would only draw a horizontal line.

Some of the students answered to (b) that lines 5 and 6 were swapped. They pointed out that the first time the `lineto` command was to be executed it was useless since the coordinates had not changed.

Interestingly, two of the students who said the program would draw a parabola answered question (b) correctly. They both wrote that, in order for the program to work properly, the variable y would have to be recomputed inside the loop. Among the students answering question (a) correctly, three also gave the correct answer to question (b) and one student had no answer to (b).

Frequency: The following table summarizes the students' answers. In total, only four of the ten students drew the program's picture correctly and five students correctly identified the problem.

	(b) correct	(b) incorrect	total
(a) correct	3	1	4
(a) incorrect	2	4	6
total	5	5	10

5.3.6 Tracing a Program [P4]

Ten students were asked to trace a given program. With this program, we wanted to directly test their ability to understand variables, functions, and assignments.

Problem: What does the following program print on the screen? Determine the program's output and briefly explain the reasoning behind your answer.

```
1 x = 5
2 def f(x):
3     return x*x
4
5 g = x*x
6 x = 8
7 print f(2)
8 print f(x)
9 print g
```

Correct solution: The program would print the three numbers 4, 64 and 25.

Students' answers: Besides the correct solution, there are four interesting types of answers to this problem.

Three students answered that the last number printed was also 64. They reasoned that g is defined as $x \cdot x$ and that x has a value of 8 at the time of the printing. From their answers it is clear that they employed a model of step-wise substitution: $g \mapsto x \cdot x \mapsto 8 \cdot 8 \mapsto 64$.

One student wrote that the statement "`x = 5`" was illegal at that position and would therefore be ignored by the computer. Hence, the output of the last statement would clearly be 64 again. Note that in this case the model of substitution is even so strong that g can be defined with no proper definition for the variable x at that point.

Another two students said that the statement "`print f(x)`" would not work at all. One thought that $f(x)$ cannot be properly evaluated as the parameter x would have to be replaced by a concrete value. The other said that, at that point, the variable x has two distinct possible values: 5 and 8. Therefore the computer cannot replace the parameter x by a concrete value since it does not know which one would be the correct one.

Finally, one student just ignored the statement "`x = 8`".

Frequency: Of the ten students only three provided a correct answer. Another four clearly used a model of mathematical substitution, arriving at an incorrect answer. The remaining three students considered the program itself to be incorrect or tacitly "corrected" it.

5.4 Discussion

Due to the small sample size, our results are not sufficient for a statistical analysis. However, the aim of this thesis is to provide a possible interpretation for the already statically proven problems students exhibit with the concepts of variable and assignment. In the section to follow, we will then further investigate possible solutions.

5.4.1 Computational Models in Mathematics and Programming

Let us briefly recall the two models underlying mathematics and programming, respectively. As we will later argue that the students' mistakes might be due to an underlying mathematical model it is important to work out the nature of this mathematical model, and its differences to the proper model for imperative programming.

Mathematics. Mathematics is primarily based on the idea of *substitution*. An equation such as $y = 2x$ establishes a relationship between the two quantities named x and y. Without further knowledge about any of these quantities, we can replace y, for instance, in another expression $5x - 3y$ to arrive at $5x - 3(2x)$, which simplifies to $-x$.

In order to substitute and simplify mathematical expressions, we can follow syntactical rules and do not need any actual values for any of the variables involved. This is for example key in solving equations. We can conduct different simplifications and transformations of the equation without knowing the value of x – even without knowing if there is a possible value for x.

In essence, mathematics uses substitution on a syntactical level to fill in terms for variables.

Programming. In contrast to mathematics, programming has variables not as mere syntactical symbols. A variable is a *reference* to a concrete object (at least during runtime) and the computations, as laid out by the program, require that each variable is a valid reference.

An assignment such as "`y = 2*x`" does not establish a relationship between two symbols. It is an actual computation, of which the result is to be stored as value of the variable y. While the values of the variables might be unknown when a program is being written, they are completely available when the program is being executed.

In essence, programming performs computations with concrete values.

5.4.2 Students' Model

A considerable part of the students does not only have difficulties with the concepts of variables and assignments, but also seems to use a mental model of the computer that is closer to that of mathematics than to how programming actually works. In particular, students attribute algebraic capabilities to the machine executing their programs.

Some students certainly use substitution. Problem [P4] gives us a direct insight into the reasoning of the students. From the students' answers, we know that four of the ten students used a model of step-wise substitution. One student even allowed the variable g to use the value of another variable x before x was properly set to a value.

What we cannot derive from the students' answers is whether they believe g to hold the expression $x \cdot x$ as its "value" or the computer to store the entire assignment as an equation $g = x \cdot x$ to later infer the value of g when needed.

Lazy evaluation. Problem [P1] about solving quadratic equations gives us fewer clues about the students' model of the machine. However, the inversion of computation and testing for computability suggests the notion of lazy evaluation, i. e., a variable's value is not computed until it is accessed.

Such lazy evaluation is consistent with a mathematical model. The assignments $x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ might, again, be seen as merely establishing a (functional) relationship between the involved quantities, giving a recipe for the later computation to be performed. When the solutions are then printed, the computer is thought to actually substitute the terms defined before and evaluate them.

Functional Relationships. Of the two graph drawing problems [P2] and [P3], the first [P2] is more interesting to our study. The second problem [P3] might strengthen our point, but it is not clear if the errors made there are due to other difficulties of the students, such as an inability to correctly read and trace a program (cf. Lister et al. [35]).

In the case of the programs intended to draw a graph, the assignment $y = \frac{x(x-5)(x+5)}{25}$ is obviously seen as establishing a functional relationship of how to compute the value of y out of the value of x. While some students could just have forgotten to recompute y when needed, it is unlikely to see such an accidental omission in several students' answers at once.

Again, the notion of a functional relationship is consistent with a mathematical model. When y is used later in the program to draw a line to a specific location, a machine following the algebraic model would first replace y by its definition and then put in the value of x.

Conclusions. Among the problems considered, [P4] clearly exposes that some students use a model of substitution as the machine's model of execution. This model of substitution can be found in mathematics and we hence surmise that students are transferring preknowledge from mathematics to programming.

The other three problems provide more indirect indication that the students are applying an incorrect mental model to the machine. However, the mistakes recorded and presented here are consistent with a mathematical model of substitution.

5.4.3 Threats to Validity

We can clearly not draw any statistical conclusions from the results obtained. Beside the small sample size, we must also be aware that the selection of high school students might not be representative of a larger population. The students who took the class had all chosen mathematics and the sciences as their elective major.

The mistakes seen in problems [P1] to [P3] might have different causes than a misconception about the underlying computational model. Not updating the value of the dependent variable in the problems about drawing a graph ([P2] and [P3]) could, for instance, be due to a simple oversight. Also note that two students in problem [P3] answered that the program would draw a parabola (instead of a line) but also clearly stated the error in the program was the missing update of the dependent variable. Hence, problem [P3] in itself would be rather weak support for our thesis.

However, the students' mistakes are consistent with a mathematical model of computation. Building on that interpretation of the students' understanding, we present a possible solution in the following sections. The relatively high success of the intervention further supports our interpretation of the results.

Finally, other studies report in part even larger numbers of students exhibiting misconceptions about variable assignments (see, e. g., Bayman and Mayer [4]). In the light of the context of such studies, we can assume that variables and assignments are difficult topics, and our findings are not just single occurrences.

5.5 Improving the Students' Understanding

In order to address the misconception about variables and the computational model, we designed a special classroom session. The concept is based on two approaches: teaching the students how to trace the variables' values through the program's execution, and explicitly confronting the students with their misconceptions.

The entire teaching session takes roughly one hour. 15 minutes are spent on an initial test, 30 minutes on the actual teaching itself with discussion and explanations, and 15 minutes for a second test, preferably taking place a day later.

Even though we did not have enough students for a statistically significant result, we found that of the 16 students taking the class, nine showed the misconception in the first test. In the second test, only one student was still consistently and two additional students were partly applying the wrong model. Hence, our teaching model seems to be a promising approach.

5.5.1 Related Work

The concept of our teaching sessions combines the ideas of *cognitive conflict* (i. e., confronting students with their misconceptions) with teaching manual tracing skills. The latter is based on the findings of a study about programmers' reading and tracing skills, which indicates that tracing the variables' values in a program increases the likelihood of correct answers. Both ideas are elaborated in the following paragraphs.

The idea of teaching students how to trace the variables' values is also used, e. g., in Hromkovič's textbook about LOGO programming [25].

Tracing skills. Prompted by the problem that many students perform poorly at solving programming tasks, a study by Lister et al. [35] investigated the students' ability to read and trace existing programs. The study's premise is that students' poor performance is not necessarily due to a lack of the ability to solve problems but rather due to fragile knowledge of the programming constructs.

Participants of the study were asked to complete twelve multiple choice questions to demonstrate their understanding of existing programming code. The questions' programs were written in Java or C++, and primarily involved arrays and loops. When the authors looked at the annotations students made on the test papers (called "doodling"), they found that some students manually traced the values of variables involved in the program, e. g., by using tables. Comparing the results, the study finds that "*if a student carefully traces through the code [. . .], thus documenting changes in variables, the likelihood of getting the correct answer is high. In contrast, not doodling only leads to the correct answer 50 % of the time*" [35].

While the study of Lister et al. is not about teaching to trace programs, its findings suggest that tracing the values of variables in the program helps students to read and understand the programs. It is therefore worthwhile to consider explicitly teaching the skill of tracing programs to students.

Cognitive conflict: confronting students with misconceptions. An important aspect of learning to program is the forming of cognitive or mental models about how a program is actually being executed. As we have pointed out before, some students seem to have and apply incorrect models about how the computing machine works.

In a study about improving the mental models of novice programmers, Ma et al. propose a teaching model based on cognitive conflict and visualizations [37]. They explain that cognitive conflict "*emphasizes the importance of explicitly challenging the pre-existing ideas held by students and thereby motivating them to form more appropriate models.*" [37].

Ma et al. also remark that cognitive conflict alone is not sufficient but must be supplemented by supporting the student to form appropriate mental models. As explained in more detail further

below, in our case we taught the students how to explicitly trace variables' values, and discussed why their mental models might not apply to programming.

Another study involving the idea of cognitive conflict is about using multimedia to teach basic principles of physics. Online multimedia treatments have become a frequently employed (additional) method in teaching. Yet it is not a priori clear how effective such multimedia treatments are and how good the learning outcome is. Muller et al. [42] have studied the learning outcomes of multimedia treatments in physics education.

Their study compared the improvement in students' understanding of physics concepts between different multimedia treatments. If the treatment included a discussion of common misconceptions, students showed a significantly better understanding afterwards. This is particularly true for students with no or little prior knowledge. While advanced students seemed to benefit less from the discussion of misconceptions it did not hurt their understanding, either. The authors conclude: "*this study suggests that raising misconceptions in traditional-style lectures should increase student conceptual understanding*" [42].

However, the study also clearly states that exposing the learner to misconceptions is not enough: "*It is important to note that although misconception-based multimedia on average resulted in greater learning gains, it is not a standalone solution to conceptual difficulties. The process of moving from alternative ideas to a coherent scientific view is complex and it remains only partially understood.*" [42]

In difference to the described study, we were not limited to a linear multimedia presentation, but had the opportunity to interact directly with the students. This allowed us to not only present a common misconception but to use an interactive approach where each student could assess for him- or herself in how far the misconception applied to him or her.

5.5.2 Teaching

Premise. The students have just been introduced to variables in Python. They know how to assign a number to a variable, how to compute a variable's value from other variables and parameters, and how to increase the value of a variable by a constant value using the operators `+=` and `*=`.

For instance, the students are able to understand and write the following program that draws a spiral.

```
s = 10
repeat 18:
    forward(s)
    left(60)
    s += 2
```

The teaching session. The sequence starts with a short test: the students are asked to manually trace a turtle program and complete the picture by labeling all lines with their respective lengths. This test is intended to make the student's beliefs and assumptions explicit and tangible (for the test, see Section 5.5.3).

As a second step, the turtle program in the test is discussed in class. At this point, the students are confronted with the correct solution and the differences to their own answers. Misconceptions become apparent at this point – particularly to each individual student.

The teacher introduces the notion of manually tracing the program and keeping track of the variables' values in a table. This is used as a tool to explain the correct solution to the initial problem, but also to discuss further examples.

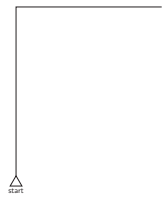
After the discussion about tracing the value of variables the teacher summarizes the important points and clearly shows in how far the concept of variable in programming differs from the mathematical concept. Key point is that a computer works strictly sequentially and stores only the numeric and immediately evaluated value of a variable. Mathematical dependencies as in $y = 2x$ cannot be expressed in programming at this stage (students did not yet learn about functions). Rather, the program would have to update the dependent variable y each time a change to x has been made.

In the class following the discussion, the students are given a second test with two programs to trace. The first program is very similar to the program of the initial test. The second program requires some transfer and application of what the students have learned. After the students have completed the tests, the correct solutions are, once again, discussed in class.

5.5.3 The Test Questions

First test. The following program draws the picture on the right. How long are the respective line segments? Label each line segment with its correct length and briefly explain your reasoning (a clear calculation suffices).

```
1 a = 5
2 b = 3*a
3 forward(a)
4 a += 3
5 forward(a)
6 right(90)
7 forward(b)
```



Solution and discussion: The first line segment actually consists of two line segments with an overall length of $5 + 8 = 13$. The second line segment has a length of $3 \cdot 5 = 15$.

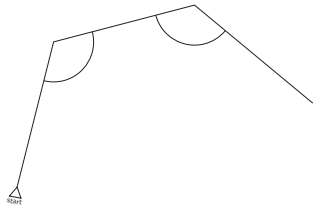
Students with a mathematical model of program execution will answer that the second line segment has a length of $3 \cdot 8 = 24$ because they evaluate b to $3 \cdot a$ first and then set $a = 8$.

The first line segment is made up of two `forward`-instructions. This is intended as a distractor from the true purpose of this test as finding the correct value of b. Students are supposed to

expect the difficulty of the problem in the two `forward`-instructions for one line segment.

Second test: Question 1. The following program draws the picture on the right. How large are the respective angles? Label each angle with its correct value and briefly explain your reasoning (a clear calculation suffices).

```
1 a = 15
2 b = a*4
3 forward(a)
4 a += 5
5 right(a)
6 forward(120)
7 right(b)
8 forward(80)
```



Solution and discussion: The first angle has a value of $160°$ (the turtle turns by $20°$). The second angle is $120°$ (the turtle turns by $4 \cdot 15° = 60°$).

Students with a mathematical model of program execution will answer that the second angle has a value of $100°$ (the turtle turns by $4 \cdot 20° = 80°$).

Since the first test used the length of line segments, we use angles' values in this second question. This involves a small transfer effort so that the students do not encounter the very same problem as before. There are two distractors in this problem. The first is the `forward(a)`-instruction. Using the variable for a forward-movement has no effect on the angle afterwards. The second problem is that the angles by which the turtle turns are not the angles asked for.

For the evaluation of the students' answers, we also accepted $20°$ and $60°$ as correct. The purpose of our study is not to test the students' geometric abilities but their understanding of variables in the context of programming.

Second test: Question 2. What figure does the following program draw? Sketch the resulting figure and label each line segment with its correct length.

```
s = 1
t = 3*s + 1
repeat 4:
    forward(t)
    left(90)
    s += 2
```

Solution and discussion: The correct answer is a square with a side length of 4.

Students with a mathematical model of program execution will answer that the program draws a spiral with increasing side lengths $4, 10, 16, 22$.

The basic design of this question is cognate with the problem of drawing a graph. However, this version requires less prerequisites and can thus be done earlier in the curriculum (drawing a graph requires the students, e. g., to be familiar with the coordinate system).

5.5.4 Results

Tests during classroom sessions. Before and after the classroom sessions, our students completed a test to assess their understanding of variables and the computational model (see Section 5.5.2). We have collected the students' answers, wrote individual feedback for each student, and kept track of how many students would exhibit indications of a mathematical model of programming.

In total, we had 16 students in our class. Two students had already extensive prior experience in programming. All other students had about 20 classes of introduction to programming with Python and turtle graphics over a period of two months.

The initial test showed strong indications of a mathematical model applied to programming. Of the 16 students, nine made the mistake of thinking that a seemingly dependent variable was automatically updated. This result is even higher than we would have anticipated from our previous results.

The second test included two problems. Both problems were answered correctly by 13 of the 16 students. We found that, in each of the two problems, only two students made the mistake of believing a variable's value would be updated automatically. It is interesting to note, however, that one student made the mistake in both problems. Two other students each answered one problem correctly and showed a misconception in the other problem.

In both tests some students obtained results that were incorrect, but could not be attributed to a misapplication of a mathematical model.

The following table summarizes the number of students giving correct or incorrect answers, respectively. As the second test included two problems, we give the numbers for both problems in the test.

Even though the number of correct answers is the same for both problems in the second test, we actually had one student who showed the misconception in both problems and two students showing the misconception in only one of the problems.

	First test		Second test		
Correct	5	31.25 %	13	13	81.25 %
Misconception	9	56.25 %	2	2	12.50 %
Other incorrect	2	12.50 %	1	1	6.25 %

Discussion. The figures and results from one class of 16 students are, again, not sufficient for a thorough statistical analysis. However, the figures presented here indicate that the proposed solution might indeed be effective in improving students' understanding of the computational model underlying programming.

5.6 Further Evidence

Based on the results obtained from the tests in classroom and the experience of the teaching session, we wanted to see how far we could reproduce the results. We created a short survey and chose two questions that were simple enough but would clearly show any misconceptions of the participant.

As this survey was addressed to students with a more advanced background, the second question included a function. In fact, this second question is as close as possible to problem [P4] in Section 5.3.

At the time of writing, we had the chance to conduct the survey in two classes of 15 and 9 high school students, respectively, in their 12th grade. None of the students attended a programming class by the author. However, the 15 students from the first class have all taken a programming class in Python about one year before the survey. The 9 students in the second class were currently attending an advanced elective course on computer science at high school.

Of the 15 students completing our survey, all 15 students exhibited the typical error in the first question. Four students also clearly used a model of substitution in the second question. Of the nine students in the second class four exhibited the error in the first question and one in the second question.

First question. What does the following program draw? Sketch the resulting shape and label all line segments with their respective lengths.

```
s = 1
t = 3*s + 1
repeat 4:
    forward(t)
    left(90)
    s += 2
```

The program actually draws a square. The definition of t, however, appears to depend on the value of s. This might be reinforced by changing the value of s – which in fact has no effect at all.

Of the 24 students who participated in the survey, 19 students drew a spiral. 15 students wrote that the lengths of the spiral's sides would be $4, 10, 16, 22$. The other four students made some minor errors in their calculations but basically arrived at the same solution.

Second question. What does the following program draw? Sketch the resulting shape and label all line segments with their respective lengths.

```
x = 5
def f(x):
    return x*x
g = x*x
x = 8
```

```
forward( f(3) )
right(90)
forward( f(x) )
right(90)
forward( g )
```

The point to look at here is the value students attribute to g on the last line. The correct value would be 25, of course.

Four of the 24 students participating in the survey did not provide an answer to this question. This might be due to the time constraints we had to impose. Of the remaining 20 students, six said that g would have a value of 64, once again reasoning that g equals $x \cdot x$ and hence $8 \cdot 8$.

Discussion. It is doubtful that indeed all 15 students of the first class would have the inappropriate mathematical model in programming. The results of the first question might therefore, in part, be due to simple oversight. The distractors such as increasing the value of s (which is without any meaning) might have been too strong, rather revealing that the students tried to solve the problem by applying structures or plans, instead of tracing the actual program (cf., e. g., Lister [34]). Yet, if the students had a strong and firm understanding of the underlying computational model, at least some should have gotten the correct answer.

The results of the second question, however, work well with our previous findings and the findings of studies such as Bayman and Mayer's [4] that about a third of the students exhibit a model of mathematical substitution in programming.

Overall, in the second class fewer students exhibited the presently studied misconception (although the number of participants is far too low to draw any statistically correct conclusions). Given that the misconception is based on students applying a mathematical model to programming, it is hardly surprising that a class with focus on mathematics and the natural sciences performed worse than a class with focus on computer science.

5.7 Conclusion

Mathematics and imperative programming share some common terminology. The underlying models and concepts, however, such as variable and function differ in key aspects. When students with mathematical training start to learn programming, they might apply the models to programming they have learnt in mathematics, leading to misconceptions and erroneous applications.

Thesis 1. *Some common misconceptions and errors made by novice programming students can be explained as the students applying a mathematical model of syntactical substitution to program execution.*

By analyzing high school students' answers to problems and programs written by these students, we found various mistakes and errors indicating deeper misconceptions. We were able to show that the errors of about a third of our students could be explained if we assume that they work with a mathematical model based on substitution. These students seemed to assume that the computational machine that executed their programs has algebraic capabilities.

Out findings also correspond well to what has been mentioned in previous studies (e. g., Bayman and Mayer [4]).

Thesis 2. *It is possible to directly improve the students' understanding and cognitive concept of variable and the computational model through explicit teaching.*

Based on preexisting teaching concepts, we designed a teaching session to directly address the misconceptions of the students about the underlying computational model of the machine. When applied in a class, the teaching session has proven successful in improving the students' understanding of variables.

Limitations and future research. Due to the small sample size of our classes, we cannot draw any firm statistical conclusions from our studies. Such a statistically significant study is beyond the scope and intention of this dissertation. The indications, however, are strong enough to warrant a larger and broader investigation of students' concept of variables and the computational model.

There is no direct way to see what mental models students construct and use in order to explain how programming works. We are required to look at indirect evidence and infer the models from the students' answers and programs. In this dissertation, we have provided a possible and consistent interpretation of the results. Moreover, students might not have a fully developed mental model of mathematics, either. The misconceptions would then be based on a mathematically inclined model, still with about the same results.

Chapter 6

Syntax Errors of Students in Python Programming

6.1 Typical Student Errors

This section presents a selection of common and noteworthy student errors we have collected. The errors are divided into three different categories (below): the first presents errors based on misconceptions about syntax and semantics. The second category lists syntax errors, which are mostly just typing errors and minor omissions. Finally, the third category is about errors which do not violate Python's grammar at all, but still give raise to various problems for students.

During classroom sessions in high school programming classes, we kept a journal with problems and misconceptions of our students. The collected notes of four years of programming classes were used as a basis for the error detection in the parser (see Chapter 7).

Several studies looked at the most frequent errors in students' programs (however, usually based on Java, e. g., [1, 12, 24, 26, 56]). In contrast to these studies, the focus of this project is rather on capturing a wide spectrum of possible errors. The actual frequencies of these errors are less important, and might be evaluated as a follow-up study.

While the following list is an attempt to capture as many errors as possible, such a list can naturally never be complete. Nevertheless, these 25 errors form the basis upon which we built our parser.

6.1.1 Misconceptions about Syntax and Semantics

A. **Invalid names.** The notion that valid names/identifiers must adhere to a set of given rules is a new concept to many students. Sometimes they choose names containing operators or even white space characters such as "`n-gon`", "`n!`" or "`paint scene`". Depending on context, we might consider the last case an instance of the "extra space"-error, instead.

   For example: `def n-gon():`

B. **Left to right assignments.** The supposed symmetry of the assignment operator has already been discussed in numerous papers and is a well known problem. Some students believe that "`x = 1`" and "`1 = x`" are both valid assignments. Particularly with more complex expressions, the students then start by writing down the expression they want to compute and add the assignment at the end, so as to save the obtained value in a variable.

   The underlying misconception might be strengthened by the use of calculators in math classes. The calculators used by our students use the syntax "$1 + 2 \cdot 3 \rightarrow x$" to store the result of an expression in a variable x. That is, the target of the assignment is in that case indeed on the right hand side.

   For example: `123 = n`

   For example: `(sqrt(p**2 - 4*q)-p) / 2 = x`

C. **Assignments in expressions.** After having learned to use augmented assignments such as "`x += 1`", some students try to use augmented assignments as a basis for more complex variable updates and embed it into an expression.

   For example: `(x *= 3) + 1`

   For example: `y = (x *= 2)`

D. **Expressions as parameters.** Parameters seem to be a difficult topic for novice programmers. Apart from problems that do not affect syntactical correctness, a few students try to modify the abstract parameters to a function instead of the arguments.

   For example:

```
def foo(2*x, y/2):
    ...
x = 123
y = 456
foo(x, y)
```

E. **Literal values instead of variables.** Similar to the use of expressions as parameters, one student directly put string literals into a for-loop where actually a variable would be required.

   For example:

```
for "e" in word:
    count += 1
```

F. **Extra or missing spaces.** Extra spaces are often found in compound names, or in the middle of a double equal sign to emphasize that two equal signs are used. Missing spaces are less common and mostly due to a typo rather than a misconception.

For example: `set pen color("blue")`

For example: `if x = = 0:`

For example: `deffoo():`

G. **Invalid else-statements.** In the beginning, many students tend to write a comparison after the "`else`". The similarity between "`if`" and "`else`" seems to give raise to the misconception that, like "`if`", "`else`" requires a condition, which then should be complementary to the one used in "`if`".

For example:

```
if x >= 0:
    ...
else x < 0:
    ...
```

A second misconception about the "`else`" is that, as the "`else`" is part of the "`if`"-statement, it must be placed inside its body.

For example:

```
if x >= 0:
    print "positive"
    else:
    print "negative"
```

H. **Missing or extra quotation marks.** Understanding the difference between a name-token and a string literal turns out to be surprisingly difficult at first. That some libraries use constants while others use strings further adds to the confusing situation (for instance, while our Python system uses string literals such as `"red"` for colors, Java uses constants such as `Color.RED`). One student even put the name of a function after the "`def`" into quotation marks since it should not be interpreted as a variable's name.

For example: `setPenColor(blue)`

For example: `def "foo"():`

I. **Division and modulo.** A quite frequent problem involves the modulo operator "`%`". On one hand, the modulo- or remainder-operation itself seems to be a difficult concept for our students. On the other hand, using the percentage-sign as an operator here seems even

more counter-intuitive. This becomes apparent when student actually use integer division when modulo is needed.

For example: `if x // 2 == 0:`

J. **Invalid use of the and-operator.** Some students start to use the `and`-operator even before it has been discussed in class and mostly just based on their intuition. These "uneducated" uses, however, lead to valid syntax but hardly ever represent the student's intention.

For example: `if a and b > 0:`

For example: `return a and b`

K. **Wrong or extra keyword.** In a few cases, students confused the keywords and used, e. g., "`for`" instead of "`if`". Other students wrote the "`def`"-keyword also when calling/invoking a function.

For example: `for x > 0:`

For example:

```
def foo(x):
    ...
def foo(123)
```

L. **Incorrect structure.** In classroom sessions, students are often asked to take an existing program and expand it. When the exercise asks for a repetition, some students just put the entire program code into a loop, including even import-statements and function definitions.

In principle, such a program as shown in the example below (using turtle graphics) works correctly. Yet, including a function definition in a loop is clear indication of a misconception in the context of novice programmers.

For example:

```
for i in range(6):
    def square(s):
        for i in range(4):
            forward(s)
            left(90)
    square(20)
    left(60)
```

6.1.2 Minor Syntactical Errors

M. **Unbalanced parentheses and brackets.** A missing closing parenthesis is a common error, but almost always a simple typing error. In Python, it is problematic only because this missing parenthesis influences all subsequent code and most Python interpreters do not recognize this error properly.

For example: `foo(12`

N. **Misspelled names or keywords.** Typing errors are very common, particularly in longer names. That modern editors highlight (correctly spelled) keywords in colors certainly helps, even though some students ignore this.

Python's dynamic nature is somewhat problematic here: misspelled identifiers are only detected during runtime when the interpreter tries to access the variable. In some cases, the error goes even completely unnoticed, introducing subtle bugs.

For example: `If x < 0:`

For example: `pint "The result is", x`

For example:

```
x = 1
for i in range(5):
    X = 2 * x
print x
```

O. **Using keywords as names.** Particularly in cases where students are not (yet) aware that a given word is a keyword, they sometimes inadvertently use it as the name for an identifier.

For example: `pass = True`

P. **Using assignment instead of comparison.** Certainly one of the most common and well known examples of a syntax error is the use of the assignment operator with a single equal sign instead of the comparison operator with a double equal sign.

For example: `if x = 0: ...`

Q. **Misspelled operators.** The misspelled operators we observed were all of the form "=*", "=!", "=+" instead of "*=", "!=", "+=" etc. In other words: the two characters making up the operator were flipped.

For example: `x =* 2`

R. **Unterminated string literal.** In some cases a string literal is missing its closing delimiter.

For example: `s = "abc`

S. **Missing colon, comma or operator.** Compound statements require a colon at the end of the first line. This colon is sometimes missing, but students usually recognize the error quickly and add the missing colon.

Slightly more interesting is the case of a missing comma or operator. We often found the missing comma in print statements. The missing operator, on the other hand, is due to algebraic notation where, e. g., $2x$ is written without the multiplication sign in between.

For example: `print 2x + 1`

For example:

```
if x != 0
    print "The result is" 1 / x
```

T. **Invalid indentation.** While indentation errors might be indication that student have not yet understood it correctly in the very beginning, we found that most errors happen rather by accident. For instance, a student might remove the loop but leave to body to be executed once and forgets to also remove the indentation of the body's code. That is why we list this problem under minor syntax errors and not under misconceptions.

For example:

```
  x = 1
while True:
    if x**3 > 1000:
        break
  x += 1
```

6.1.3 Beyond Syntax Errors

U. **Code after a break-statement.** It is not always properly understood that a break- or return-statement means that the loop or function, respectively, is left immediately. A break-statement is sometimes followed by further code to be executed after the loop has been left. In other cases the return-statement is seen as merely returning the value and followed by a "`break`" so to actually leave the function.

For example:

```
t = 2
while t < x:
    if x % t == 0:
        break
        print "Not a prime"
    t += 1
```

For example:

```
def foo(x):
    return x**2
    break
```

V. **Call without parentheses.** Novice programmers are sometimes puzzled as why their functions are not executed. Investigating their programs reveals that they either did not try to call the function at all or forgot to put (empty) parentheses after the function's name.

For example:

```
def foo():
    for i in range(10):
        print i**2
foo
```

W. **Useless computations.** Before actually using a function in an expression, some students would first call that function just prior to its actual use. When asked about it, they explained that first they have the result be computed and in the subsequent statement(s) this result is then actually used.

For example:

```
sqrt(x)
if 0 < sqrt(x) < 1:
    ...
```

X. **Useless Comparison.** Python supports testing against a range, e. g., "`if 0 < x < 1`". Particularly when negative numbers are involved, however, the test might not be satisfiable, leading to errors which are hard to detect.

For example: `if -3 <= x < -12: ...`

We found one student in a programming class that used "Processing" (a programming language based on Java) to consistently compare everything with `True` and `False`. In Python, this does not work at all.

For example: `if x < 3 == True: ...`

Y. **Shadowing.** Students surprisingly often reuse names for different purposes in their program. We give here just two examples of this problem.

For instance, during a class about turtle graphics, a student used the name of a function again as a parameter to his own function (see below). When the program is executed, Python complains that it cannot call an integer. The problem, of course, is not the intention to actually call an integer but that the parameter shadows the function, which was intended.

In another case, students reused the name of a decorator (annotation) to a function for the function itself. While this works in simple cases, it, once again, leads to bugs later on, when another function should receive the same decorator.

For example:

```
def move(left):
    forward(10)
    if left == "Yes":
        left(90)
    else:
        right(90)
```

For example:

```
@memoize
def memoize(x):
    ...
```

6.2 Related Work

How well does our list as given above cover actual errors? Are there any related studies we could use as a reference? Unfortunately, to our knowledge, there are no studies about errors in Python. A direct comparison and assesment is therefore difficult, and a comparison to errors in other programming languages is of limited value.

Nevertheless, since most recent studies focus on Java, we compare our results to the results obtained by these studies. We have to keep in mind, though, that there are significant differences between Java and Python, which must be taken into account.

6.2.1 Errors in Java

Numerous studies have investigated the frequencies of different errors in novices' Java programs. As there are no comparable studies in Python, we use three of these studies [1, 24, 56] as a basic guide to determine if our list of syntax errors is missing important problems. However, many of the more frequent syntax errors in Java cannot occur in Python, anyway.

Some of the problems reported in Java are caused, e. g., by misplaced semicolons. Students either forget to put a semicolon at the end of the statement, or, often even worse, put a semicolon after the header of an if-statement, say. The problem of such an extra semicolon is that it is syntactically legal but results in an empty body – something which is hardly ever intended by a novice programmer.

Python has semicolons as well. They can separate or conclude small statements on a single line (see section 3.4). There is virtually no need, however, to put several statements on one line, particularly not in the context of novice programmers. Finally, an empty statement must explicitly be written as the `pass`-statement. Hence, a novice programmer simply cannot inadvertently create a compound statement with an empty body. (This does not mean, of course, that Python is a better language than Java, but simply that the errors reported concerning semicolons cannot occur in Python).

Java-errors that translate to Python. The following table shows those Java problems, which we consider directly applicable to Python as well. For each of these problems, we have given the letter identifying the error in our list of student's errors. Three entries have no direct correspondence, as they are runtime errors in Python, and are therefore not handled by the parser at all.

= versus ==, e. g. `if (x = 7)`	P
mismatched brackets or string delimiters	M, R
using keywords as identifiers	O
forgetting parentheses after method call	V
>= versus =<	Q
not using return value of function	W
confusing parameters and arguments	D
use of comparison after &&, e. g. `if (age >= 13 && <= 17)`	J
`else` without `if`	G
misspelt word or command	N
missing identifier, e. g. `public class { ...`	(A)
cannot find identifier	–
invoking methods with wrong arguments	–
package does not exist	–

Java errors not applicable to Python. Due to the large differences of syntax and semantics between Java and Python, not all errors typically found in a Java program can occur in a Python program. Some examples of inapplicable Java problems are given below.

Two of the problems could be translated to Python but we feel that the difference is still too large as to consider them *directly* applicable to both languages. Since `and` and & differ much more in Python than their Java counterparts && and &, a confusion is less likely.

&& versus &	`and` versus &
missing braces	missing indentation
flow reaches end of non-void method	–
== versus `.equals`	–
incorrect semicolon resulting in empty body	–
wrong separators in `for`-loops	–
loss of precision, e. g., `int i = 7.3;`	–
incorrect semicolon after method header	–
improper casting	–
duplicate variable in the same scope	–

6.2.2 Conclusion

Our list of errors seems comprehensive enough to include frequent errors reported from other studies about novice programmers.

Chapter 7

Parsing Python-Programs of Novice Programmers

7.1 Introduction

Python has found widespread use as an educational programming language. While Python's simplicity is well suited for novice programmers, the compiler's feedback certainly is not. Python's reference implementation uses more or less a single error message: "*syntax error*" with no further information. It might even be displayed at the wrong position. But: are these scarce error messages inherent to Python? Is Python as a language limited to crude error reporting or could a compiler provide the necessary feedback for novice programmers?

This chapter presents the techniques used to build a Python parser, capable of detecting and handling the collection of student's errors presented in Chapter 6. At its core, the parser is based on a LL(1)-parser, which has been enhanced to better detect syntax errors, resulting in an LL(*)-parser. In order to recognize errors as accurately as possible, the parser considers the entire statements, and not just individual symbols. A line starting with "`wile`" and ending in a colon, for instance, can thus be recognized as a misspelled **`while`**-statement.

The parser as described in this chapter is part of an educational Python environment *TigerJython*. It is already in widespread use, particularly in high schools in Switzerland and Germany.

7.1.1 Organization

Section 7.3 gives an overview of the three stages of the parsing process, that includes the *lexer*, the *preparser*, and the actual *parser*, which are discussed in more detail in Section 7.4, and Section 7.6, respectively.

Of particular importance is the handling of brackets and parentheses, as discussed in Section 7.5. Brackets are both a frequent source of syntax errors, and an important structuring element.

Finally, Section 7.7 discusses to what extent static analysis (e. g., determining an identifier's type) is possible in Python, and how it is actually used to recognize errors.

7.2 Premise

Let us start with a few assumptions about the programs we want to parse.

- The programs are small enough, so that their representations fit entirely into memory,
- The programs usually contain (syntactical) errors,
- The syntax errors are isolated, i. e., there is not more than one error per statement.

The first point is to say that we do not concern ourselves with efficiency, or the problem of how to compile large projects that do not fit into memory. After all, our target audience are novice programmers and educators, not professional software engineers.

Second, there are already great parsing techniques for syntactically correct programs. For Python, you could simply use the parser generator of your choice, feed in Python's grammar and be done. However, *error handling* of existing Python parsers does not meet our requirements for an educational system.

Finally, we put some restriction on the number and kind of errors we permit. If the given input lacks basic structure or enough information to reconstruct a valid Python program, our parser is allowed to simply give up. Section 7.2.1 gives more details about the restrictions we impose on input programs.

Restricting Python's grammar. Some of Python's features clearly address the advanced programmer, and are not used in introductory classes, or by novice programmers. In principle, we could therefore choose a subset of Python, and restrict the parser to accepting just programs in that subset.

We decided, however, to rather build a parser that accepts all syntactically legal Python programs. This also implies, for instance, that a function's name without parentheses remains a syntactically legal statement – even though it has no side effects. In addition, forgetting to put the parentheses to call a function is a rather common novice error. Instead of just executing a program that, in the end, does nothing, we would like to provide an error message, pointing out that the parentheses are missing.

We resolved this problem by introduction of configurable extended error checking in the parser. By setting a flag, the parser can be operated in «strict mode», and thereby disallow statements without side effects, for instance.

Not restricting the parser to a subset of Python also means that it can parse Python libraries, which might be imported into a student's program. This is important for static analysis.

Related goals. There are a number of tools to check a Python program for possible problems and errors. To our knowledge, however, all of these tools require a valid abstract syntac tree (AST) of the program to operate on. Thus, in case of *syntax errors*, the tools are of no help to specify the exact cause of the error. In contrast, this thesis' main concern is with creating an AST for a given (erroneous) input Python program.

Another interesting problem is the correctness of a computer program. Again, checking the correctness of the input program is far beyond the scope of this project. Whenever we speak of correctness, we mean *syntactical correctness*, and hence the possibility to create a proper AST for the given source code.

7.2.1 Isolated Occurrence of Errors

A key requirement for our parser is that all syntax errors occur "isolated", i. e., we can think of the program as correct, except for one single syntax error. The intention behind this requirement is that, in case of a syntax error, the parser can look at the error's environment, and infer the exact cause for the error. If there is enough information in the error's surrounding environment, the parser might even be able to fix it, and continue parsing the program.

For instance, the parser might encounter the sequence of the four tokens $\langle dfe \rangle \langle foo \rangle \langle (\rangle \langle) \rangle$. The juxtaposition of two names is certainly a syntax error, but what is the actual mistake? Here are some possibilities (along with the corrected forms), from which the parser has to pick one:

- a missing assignment, i. e., `dfe = foo()`,
- an extra space, i. e., `dfefoo()`,
- a misspelled keyword, i. e., `def foo()`,
- a missing comma or separator, i. e., `dfe, foo()`.

Now, the above sequence "`dfe foo()`" might be embedded into a structure as follows:

```
dfe foo():
    pass
```

By looking at this structure the parser can now infer that the most likely source of the error is a misspelled "`def`"-statement (in a nutshell, any other hypothesis would require more changes to yield a correct program, since only the `def`-statement makes sense for the suite following it).

For such an analysis, however, we must assume that the direct environment of the syntax error is correct and does not contain further syntax errors.

Hence, any program sent to the parser is assumed to fulfill the **isolation requirements for syntax errors**:

1. **For every syntax error, one can take out a connected part of the entire program so that this part is a valid program and contains exactly this and only this one syntax error.**

2. **In the given input program, any given name is misspelled at most once.**

Python's program structure certainly facilitates the first requirement since almost every statement in itself already forms a valid Python program. In many cases, we can thus read this requirement as there being only one syntax error per line.

The second requirement means that we can look at the rest of the program to decide whether "`deffoo`" or "`dfe`" are used anywhere else as proper names. This would not work if, e. g., "`dfefoo`" was misspelled multiple times as "`dfe foo`". In short: any identifier appearing more than once is assumed to be correct.

In practice, it turns out that the parser is often capable of handling a larger density of errors. But we do not *require* the parser to correctly identify and handle errors in such cases.

7.2.2 Standard Python

Python traditionally puts little emphasis on the handling of syntax errors. Apart from indentation errors, a syntax error is usually reported whenever Python cannot continue the parsing process, highlighting the "offending" symbol that defies parsing. In various cases, however, the true problem is not at the highlighted symbol, but located before.

We discuss an example program to illustrate how far typical error reporting can be off. The actual error here is in line 1 with the missing closing parenthesis after the call to `foo`. Seen from the perspective of Python's grammar, however, the error is found (and in fact reported) in line 3 with the call to "`spam`". Depending on the actual structure of the program, the discrepancy between the actual error and the reported position can be almost arbitrarily far.

```
x = foo(
y = bar()
spam(x+y)
```

Without the closing parenthesis in line 1, Python's parser interprets this code segment as follows (the actually reported error can be seen in Fig. 7.1.):

```
x = foo(y = bar() spam(x+y)
```

In other words, the assignment to `y` is interpreted as a named argument to the function `foo`. Even in this interpretation, the true error would be a missing comma or operator in front of `spam`. Anyway, there is nothing inherently wrong about the symbol `spam`.

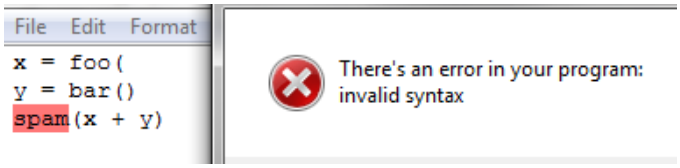


Figure 7.1: Python 2.7.11 reports that the symbol "`spam`" is the source of the syntax error.

Python's error reports are not only minimalistic, but can also be misleading. How should a novice programmer recognize that, despite that fact that `spam` was reported as erroneous, the actual problem and source of the error lies elsewhere?

7.3 The Parsing Process

7.3.1 Overview

The process of parsing commonly involves two steps: the lexer accumulates individual characters to form tokens. The parser then consumes these tokens and creates the abstract syntax tree (AST) from it (Fig. 7.2).

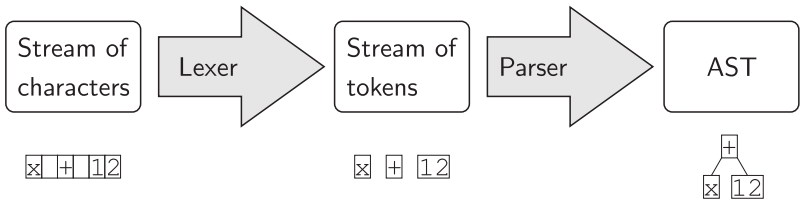


Figure 7.2: The parsing process typically involves a lexer and the actual parser to transform the input stream of characters into an abstract syntax tree (AST).

The parser presented here deviates from this model by an additional stage inserted between the lexer and the actual parser. A *preparser* transforms the stream of tokens coming from the lexer and presents the parser with a tree of logical lines (Fig. 7.3, recall that a single *logical* line might be distributed among several actual or *physical* lines). The tree's structure actually represents the indentation levels of the lines. For instance, the lines that make up the body of a compound statement (e. g., a loop) become child nodes to the compound statement's node.

Error Handling. Error handling is distributed among the different stages. The lexer, the preparser, and the parser are all responsible for detecting, reporting and handling syntax errors.

On the level of the *lexer*, the isolation requirement for syntax errors allows that any single character could have been deleted, inserted, changed, or swapped with its immediate neighbor.

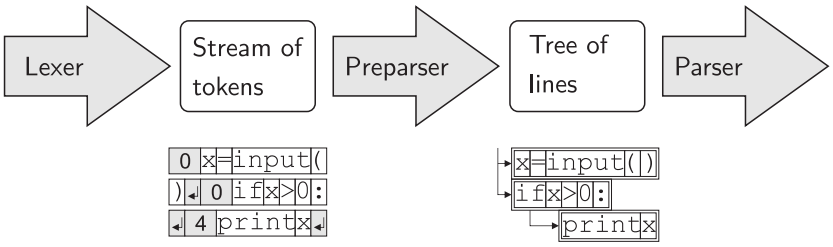


Figure 7.3: Between the lexer and the parser, there is an additional *preparser* that transforms the sequence of tokens into a tree of logical lines. Note, how the character stream on the left stil lcontains ⟨*NEWLINE*⟩- and indentation-tokens, whereas the tree of logical lines on the right has no need for these tokens anymore.

There are, however, only very few cases where the lexer can determine that a syntax error is present, find its cause and source, and fix it. For instance, because of lacking context, the lexer could not decide if a word such as "`dfe`" is meant to be an identifier or a misspelled keyword "**`def`**". The lexer can, however, detect and fix, e. g., unterminated strings.

In order to build logical lines from the token stream, the *preparser* needs to detect and handle indentation problems, as well as unmatched brackets and parentheses.

For the *parser*, the isolation requirement for syntax errors basically means that only a single token per line might be wrong. This includes cases such as a deleted or inserted token, or even swapped tokens, when directly inherited from swapped characters. The parser's job is to identify the exact syntax error or erroneous token, and, if possible, to correct it.

The purpose of correcting syntax errors is not, of course, to execute a faulty program. The idea is rather that, if the parser can correct a program, it can provide meaningful messages to the user. In addition, the abstract syntax tree (AST) is used to construct a symbol table, which in turn is used to help identifying other syntax errors.

7.3.2 Structure of Python Programs

The primary element to structure a Python program is indentation. According to the Python specification [48], the lexer is supposed to keep track of current indentation levels and then generate ⟨*INDENT*⟩- and ⟨*DEDENT*⟩-tokens to mark the beginning and end of code blocks ("suites" in Python's terminology, cf. Section 3.4). Inside brackets and parentheses, however, all line breaks – and accordingly also indentation – are ignored.

Now, consider the case of a missing closing/right parenthesis or bracket. This leaves the lexer in the state of ignoring the primary structuring element for the rest of the program. As seen from the parser, that entire rest seems to occur on a single line (cf. Section 7.5). Finding a good point to fix the problem and insert the missing bracket becomes virtually impossible for the parser.

Throwing away information so crucial as the indentation at that early stage is not a viable option. The lexer must keep this information until such time as it could establish that all brackets and parentheses were properly closed/matched.

Indeed, we have the lexer retain all line breaks and indentation. In addition, it also keeps track of parentheses and brackets during the scanning process. In case of any unbalanced bracket, the lexer sets a flag indicating that bracketing information is not reliable and must be fixed. The preparser between the lexer and the parser reads that flag and tries to reconstruct the correct structure of the program.

In the end, we want to pass on information about the program's structure from the lexer to the parser. In particular, we want to present the parser directly with logical lines instead of the ⟨*INDENT*⟩- and ⟨*DEDENT*⟩-tokens typically used in Python. The power of logical lines stems from the fact that each logical line usually contains one statement. In addition, a logical line is self-containing and can be completely parsed and executed on its own (this fact is used, e. g., for the interactive console/REPL). That is, because statements are separated by line breaks, we can quite reliably split the program into individual statements and have the parser operate on these statements. Should the parsing of any one logical line fail, we get recovery for free, and the parser can just continue on the next logical line/statement in the list.

In order to implement the idea of logical lines as a basis for the parser, we insert a *preparser* between the lexer and the parser. It takes the stream of tokens from the lexer, analyzes its structure and creates a tree with Python's logical lines as nodes. Each node has an associated list of the tokens that make up the logical line, and can have a list of child nodes in case the statement has a body (Fig. 7.4). Note that some statements, such as `if`/`else`, will still span more than one logical line.

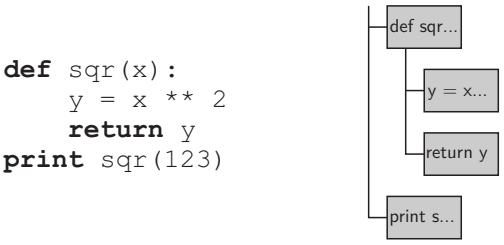


Figure 7.4: Before parsing, a Python program is transformed into a tree with logical lines as its nodes.

The parser is in a very comfortable situation. It can now look at each logical line individually. Fixing errors becomes a local task. Recall that we still assume that each individual line might contain a syntax error. But as we deal with closed entities, we can use simple pattern matching and include tests such as "has a colon at the end" or "has a suite".

7.3.3 Drawbacks of this Process: Changing the Grammar

Print: function versus statement. While "`print`" is a keyword for the print-statement in Python 2, it has become a mere function and hence a name in Python 3. Python 2, however, supports a special statement to change the grammar and make "`print`" a function:

```
from __future__ import print_function
```

For one thing, this statement must be executed by the parser and not by the runtime system. For another, at the time the parser encounters the statement, the lexer has already created ⟨*print*⟩-tokens, which must then all be replaced by respective ⟨*name*⟩-tokens.

Another alternative to approach this is, of course, that the lexer itself looks out for future-imports at the beginning of the module.

7.4 The Lexer

Most modern programming languages share a common and established set of lexical rules with only slight variations. Python is no exception here, and creating tokens from the input characters is a straightforward task.

In accordance with the Python specification, the lexer reports an error in case of invalid input characters (in a German speaking region this is of some importance as non-Ascii characters are frequently used). In a number of additional cases, the lexer can also report an error, fix the problem and still generate valid tokens. The two interesting cases are string literals and multi-character operators, as discussed further below.

7.4.1 Symbol Table and Brackets

The lexer not only tokenizes the input program. It also keeps tally of all names/identifiers in the program and checks the input for possible problems regarding brackets and parentheses.

Symbol table. During its tokenizing process, the lexer creates a simple symbol table, keeping track of how often any name occurs within the source program. This information is later accessed and used by the parser to handle misspelled names and keywords.

Consider, for example, two adjacent identifiers such as "`first value`". Two identifiers can never stand together, so this is clearly a syntax error. However, the parser has various alternatives how to fix it:

(A) form a single word: `firstvalue` or `firstValue`

(B) insert an underline: `first_value`

(C) insert a dot: `first.value`

(D) insert a comma: `first, value`

(E) insert an operator: `first + value`

In general, the parser cannot distinguish between variants (D) and (E) (apart, of course, from some special cases). However, using the lexer's symbol table the parser can see how often the names "`first`", "`value`" and "`firstvalue`" occur in the source program and base its decision for an alternative on these frequencies. This statistical approach does not always work, though, and might even choose the wrong alternative.

The advantage of this symbol table in the lexer comes from the fact that the lexer has already scanned the entire source code before the parser starts its work. Hence, the symbol table provides a global view with respect to the occurring names. On the other hand, imports are not taken into consideration. This means that the lexer's symbol table does not suffice for performing a statical analysis of names.

Brackets. In addition to the symbol table, the lexer keeps track of parentheses and brackets using a stack. This way, unmatched brackets are already discovered during the tokenization of the source program. The lexer, however, does not attempt to fix any problem with brackets but sets a flag indicating that brackets are not balanced. Later on, the preparser will take on the responsibility to identify and, if possible, fix bracket-related problems. A more detailed discussion follows in Section 7.5.

7.4.2 String Literals

Like parentheses, string literals might be missing the proper closing or terminating character. In difference to parentheses, this is quickly detected because a string literal is not allowed to contain line breaks (except, of course, for multi-line string literals in triple quotation marks). The question is where to insert the missing closing delimiter.

In general, it is not possible to reliably find the exact extent of the string literal if one of the delimiters is missing. The error is, however, clear enough so that even novice programmers quickly know how to fix it – especially with the aid of modern editors, which highlight string literals in color. Still, we can do a little better than just insert the missing delimiter at the end of the line.

String literals are very often part of an expression, i. e., they are embedded in function calls (as arguments) or part of a list or dictionary (in Python's standard library, string literals are immediately followed by a closing bracket in $21.8\,\%$ of 40 777 cases). This is important with respect to our goal of producing a sequence of valid tokens. The lexer should insert the closing delimiter so that, ideally, the program can be parsed without further syntax errors.

For instance, consider the example

```
if s.startsWith("abc):
    pass
```

If the lexer places the string delimiter at the end of the line, the parser will later on report a missing parenthesis and colon – even though they are actually present.

Before placing the closing string delimiter, the lexer checks for a colon and any closing parentheses or brackets that would match the current bracket stack. With relative ease, the lexer can thus restore basic structural information.

7.4.3 Operators

Compared to keywords and names, operators tolerate very little errors before they become unrecognizable. In other words: there are only very few cases where the lexer can fix a syntactically incorrect operator symbol. Fortunately, these cases include some of the errors students are more likely to make.

During our classroom sessions, we observed that some students put a space in the middle of operators with two characters, as, e. g., the double equal sign "==" used for comparisons. A second error we could observe quite frequently was the swapping of the two characters and writing, e. g., "=*" instead of "*=". While the lexer can easily detect two single equal signs separated by a white space and remove the extra white space, the second case is harder to correct. The problem arises when "x=-1" might be either "`x -= 1`" or "`x = -1`".

In fact, we decided to only correct clearly invalid cases, i. e., the lexer interprets even "`x =- 1`" with an extra space between the minus and the digit as the assignment of -1 to x. The only violation of this rule is "=*". The star could belong to the following expression as an unpacking operator. But if the expression is a number, there is nothing to unpack and the syntax would be illegal. Hence, "`x =* 2`" is corrected to "`x *= 2`".

7.5 Brackets and Parentheses

Brackets and parentheses not only play a crucial role during parsing but are also a frequent source of errors. One of the most common syntax errors in that regard is a wrong number of closing parentheses, including having none when one is actually required. Particularly in case of nested function calls or complex mathematical formulae, closing all parentheses correctly is difficult.

The importance of brackets and parentheses stems from the fact that they directly influence the structure of the source code. Brackets even establish contexts relevant for parsing. In Python, for instance, statements cannot occur inside brackets, and symbols such as the assignment operator '=' or the colon ':' change in fact their meaning. In particular, line breaks and indentation are ignored inside brackets while playing a pivotal role outside to structure the program code.

Simply detecting bracket-related syntax errors seems easy at first: we just need to keep track of currently open brackets, for instance using a stack, and check whether a closing bracket matches the opening bracket on top of the stack. The problem arises if we want to correct missing or mismatched brackets, or even determine the exact kind of error: in many cases it can be very hard to determine if, for instance, a closing bracket does not match the opening counterpart because the opening bracket is missing or swapped with another closing bracket.

The parser uses line breaks as synchronizing tokens after an error. Hence, if the parser cannot recover while parsing a given statement, it resumes parsing the subsequent statement. Due to the fact that new lines are ignored inside brackets and parentheses, the parser must fix bracketing problems, and reconstruct the actual structure of the program code as well as possible or abandon further parsing altogether.

7.5.1 Goals

Ideally, the parser would be able to point out the exact location where a missing bracket must be inserted or an extra bracket deleted. It would also correctly report if a bracket is in the wrong place and find the correct location for it. Due to the inherent ambiguity of erroneous code, however, there is always an uncertainty about the presented solution to fix the problem.

When looking at data of novice programmers' mistakes, we find that a missing closing parenthesis is among the most common syntax errors (e. g., [1, 12, 24, 26, 56], – the actual data is for Java, but this error directly applies to Python as well). As shown above (Section 7.2.2), this error is also poorly reported by standard Python. The first goal is hence that the parser detects all missing closing parentheses, and either finds a reasonable and good choice where to put the missing closing parenthesis, or reports that the opening parenthesis has no matching counterpart.

As the brackets and parentheses highly influence a program's structure, the parser should be capable of restoring the basic structure even in the case of malformed bracketing. That is, after un- or mismatched brackets, it must recover as soon as possible to continue the parsing of the remaining program code in a meaningful way.

7.5.2 Combinations of Brackets and Other Tokens

We already know that some combinations of tokens are invalid. For instance, an opening curly brace can never follow a name token. On the other hand, parentheses very often follow a name token. Hence, a statistical analysis of how brackets and parentheses are actually used might help to decide, in case of a syntax error, which of various options is more likely to be the correct one.

We used Python's standard library to accumulate the information about the usage of parentheses. The rationale behind this choice is that each Python module is also a valid program, and that there are currently no large collections of samples from novice programmers available. In

total, 205 files were analyzed, making up around 4 MB of text data. For each bracket token, the immediately preceding and the immediately following tokens were counted.

Results. With almost 70 000 opening and closing parentheses, they certainly make up the largest group. Interestingly, in more than 90 % of all cases, the opening parenthesis directly follows a name, i. e., constituting a call. This is further supported by noting that closing parentheses are much more likely to be directly adjacent to another closing parenthesis than their opening counterparts (7 % vs. 1 %), indicating structures such as, e. g., "`foo(bar())`". Around 14 % of the parentheses are actually empty and half of the closing parentheses are followed by a linebreak.

Square brackets are still very likely to directly follow a name with more than 70 %. In total, square brackets are used in about 80 % of the cases to access a sequence's element with the remaining 20 % to directly define a list. Since indexed access cannot be empty, we find that half of all lists directly defined in Python are actually empty.

Curly braces are much less frequent with less than 1000 occurrences. This is, however, not surprising when compared to the fact that the large bulk of used brackets and parentheses constitute in calling and accessing elements while curly braces are only used to define literal dictionaries and sets. Of these, even 75 % are actually empty. For later uses, it is also noteworthy that curly braces are frequently enclosed by parentheses but that in less than 1 % does a parenthesis occur directly inside curly braces.

Each table below lists the four most frequent tokens preceding and following an opening or closing parenthesis or bracket, respectively. Reading example: the token in front of a opening parenthesis is in 90.5 % of all cases a ⟨*name*⟩ and in 1.5 % the modulo-operator ⟨*%*⟩. The closing parenthesis is only preceded by a ⟨*name*⟩ in 54.5 % and followed by a ⟨*newline*⟩ (abbreviated in the table as ⟨*nl*⟩) in 52.5 % of all cases.

Parentheses. $N = 68352$

⟨*name*⟩	`%`	`=`	`(`	`(`	⟨*name*⟩	`)`	⟨*str*⟩	⟨*int*⟩
90.5 %	1.5 %	1.1 %	1.0 %		66.8 %	14.1 %	12.5 %	3.0 %
⟨*name*⟩	`(`	⟨*str*⟩	`)`	`)`	⟨*nl*⟩	`:`	`)`	`,`
54.5 %	14.1 %	11.3 %	7.1 %		52.2 %	27.7 %	7.1 %	3.5 %

Square brackets. $N = 13472$

⟨*name*⟩	`=`	`)`	`]`	`[`	⟨*name*⟩	⟨*int*⟩	`]`	⟨*str*⟩
72.5 %	15.8 %	4.2 %	2.0 %		38.8 %	26.0 %	10.0 %	9.6 %
⟨*int*⟩	⟨*name*⟩	`[`	`:`	`]`	⟨*nl*⟩	`=`	`)`	`,`
33.3 %	32.5 %	10.0 %	9.6 %		40.9 %	14.3 %	12.3 %	8.0 %

Curly braces. $N = 981$

=	,	(	:	{	}	⟨*nl*⟩	⟨*str*⟩	⟨*name*⟩
83.3 %	7.7 %	3.5 %	1.6 %		75.3 %	12.0 %	10.4 %	1.0 %
{	⟨*nl*⟩	⟨*str*⟩	⟨*name*⟩	}	⟨*nl*⟩	)	,	[
75.3 %	12.8 %	4.1 %	3.5 %		80.2 %	9.4 %	9.0 %	0.8 %

7.5.3 Detecting Errors

The lexer keeps track of brackets and parentheses while scanning the source program. Using a stack, it detects if a closing bracket does not match the current opening bracket. After having scanned the entire document, it then checks whether the stack is empty, that is, all brackets have been properly closed. If any mismatch is detected, the lexer stops keeping track of brackets and sets a flag to indicate that the bracketing is erroneous.

There is a dedicated phase in the parsing process to fix brackets and restore the structure of the source code. This phase is only applied in case the lexer has found mismatched brackets.

The "bracket fixing"-phase of the parser first creates a list of all brackets and parentheses in the source program. The algorithm then searches for matching brackets in the list and removes them, analogous to how we would resolve parentheses in mathematical formulae. Once the parser has no further matching pairs to remove, it investigates the remaining patterns, tries to resolve mismatches and resumes removing matching pairs.

Swapped brackets. We first discuss the case of both opening and closing brackets remaining. Then there is a first closing bracket, which does not match a preceding opening bracket. By looking at the surrounding brackets it is possible to detect the case of two brackets having been swapped, as, for instance, the two closing brackets in "`(...[...)]`". In order to be recognized as swapped brackets, the two tokens must be directly adjacent to each other.

There are several heuristics to guide the parser in cases where both the opening and the closing brackets could have been swapped.

- Curly braces never follow a name whereas parentheses and square brackets are often preceded by name.
- We assume that lists and tuples tend to be homogeneous. If the contents of the outer brackets (ranging from the first opening to the second closing bracket) is part of a list or tuple, we can check if other elements are in brackets or parentheses as well and try to match them.
- It is unlikely for curly braces to contain one single further expression in parentheses. Curly braces are often used for dictionaries – in which case the contents cannot be completely

in parentheses or brackets. Hence, an expression of the form `{(...)}` would have to be a set with exactly one entry. Even though this is just a statistical guess, the parser swaps brackets in such a way that curly braces end up at the inside with other parentheses or brackets surrounding them.

If, eventually, the parser still cannot decide and statistics is no help, it swaps the closing brackets. Here, we just assume that a programmer puts more care on opening brackets than on closing them.

Mismatched brackets. When swapping brackets fails, the parser must look at the two mismatched brackets in isolation. It still takes the surroundings into account and, by counting opening and closing brackets, it makes sure that the problem is indeed one of mismatched brackets and not one of an extra or missing bracket in the middle (discussion in the next paragraph below).

Mismatched parentheses and brackets are resolved by assimilating the two brackets. We assume that either the opening or the closing bracket is correct and the respective other needs to be adapted. The parser's problem is to decide which one of the two brackets is the correct one.

Analogous to the problem of swapped brackets, the parser uses some heuristics and rules derived from the grammar to decide for one of the available options.

- As before, curly braces cannot follow a name, while parentheses and square brackets often do. Thus, if the opening bracket is preceded by a name, the parser can rule out curly braces.
- In contrast to curly braces, square brackets and parentheses cannot contain colons. To test this, however, care must be taken because the brackets in question might enclose a valid dictionary.
- The statements `def` and `class` require parentheses after the name. The statement `del`, on the other hand, can be used to delete either variables or elements of a sequence, for which square brackets are used.
- If the brackets span everything between an `if` or `while` until a colon, they must be parentheses enclosing the test expression.
- The operator `in` is most likely followed by a list, which uses square brackets.

Single brackets. There is some extra information available if the single opening or closing bracket is enclosed in surrounding, matching brackets or parentheses. In that case, the parser can limit the search for a possible matching location to the interior of the surrounding brackets.

As long as the tokens are surrounded by other brackets, finding the correct action to repair the program code is not as important. Independent of the chosen option, line breaks and indentation is still masked between the surrounding brackets.

For a missing closing bracket, this can be as simple as inserting the missing bracket in front of the surrounding closing bracket. The parser has the tendency to insert a missing closing bracket and delete an extra closing bracket.

To find the location for a missing opening square bracket or parenthesis, the parser looks for an invalid combination of two tokens that would allow for the opening bracket to be inserted. For instance, two name tokens cannot be directly adjacent to each other in Python. However, name tokens are often followed by square brackets or parentheses and can act as first token in an expression.

7.6 The Parser

Starting point for our implementation is a simple LL(1)-parser based on Python's grammar. Implementing this LL(1)-parser by hand is a laborious task, but later on it allows us to modify it at will and include cases for error recognition and correction. We use Python's own standard library as a test suite for the parser before we start to make any modifications.

Even our very simple implementation is capable of reporting a few errors with more accuracy than just "syntax error". Whenever we expect a very specific symbol such as the colon at the end of "`if`" and "`def`", the parser can report an error of the form: "⟨*token1*⟩ *required but* ⟨*token2*⟩ *found*". In case of any other error it reports a "*no viable alternative at* ⟨*token*⟩" (this is more or less what *Jython* offers out of the box).

The goal of all subsequent modifications is to replace the generic messages by more specific error messages. In particular, the "*no viable alternative at* ⟨*token*⟩" should be completely eliminated since it does little to help a novice correct the program.

7.6.1 Recognizing Structural Errors

The parser receives its input in form of a token tree. The preparser makes sure that each logical line is represented by a node in the tree so that the parser can basically parse each logical line separately. The exception to that rule are compound statements with multiple branches. An if-else-statements spans two logical lines and hence two nodes: one for the if-part and one for the else-part.

Else without if. In a list of nodes (logical lines), the parser first looks for statements spanning more than one line, most notably else-statements, and tries to group these nodes for further parsing. This becomes interesting once the parser detects an "*else without if*".

Placing the "**else**" at the correct position is difficult for novices. We first noticed this problem in an earlier Java class (we used Java as a teaching language before switching to Python). A student wrote:

```
if (x >= 0) {
  println(math.sqrt(x));
  else
  println("There is no square root");
}
```

This pattern later occurred in Python classes as well:

```
if x >= 0:
    print math.sqrt(x)
    else:
    print "There is no square root"
```

When asked about their code, the students explained that since the "**else**" is part of the "**if**"-statement it must be placed into its body.

Reporting the error "*else without if*" is confusing in such a situation because the "**if**" is clearly present. In Python's terminology the problem is not a missing "**if**" but a wrong indentation of the "**else**".

How should the parser detect and react to a lonely "**else**" then? It must check if the parent-node in the tree is an if-statement and report that the "**else**" must be dedented to match the "**if**".

Else with condition. A second misconception about "**else**" the parser must watch out for is the notion that the "**else**" requires or allows for a condition just as "**if**" itself (this might be a problem of non-native speakers, though, who are not familiar with "else" as an English word). Some students even had several "**else**"-statements for a single "**if**" to form a multi-branch. By giving specific error messages the parser is able to help students correct their programs and learn how conditional statements must be formed correctly at the same time.

The idea of multiple "**else**"-branches attached to a single "**if**" might look as follows. Here, each "**else**" also received a condition to test for on its own.

```
if x > 0:
    ...
else x == 0:
    ...
else x < 0:
    ...
```

In such a case, we expect the parser to recognize the condition given between the "**else**" and the colon and point out that "**else**" does not allow for such a condition to be placed after the "**else**". Ideally, the parser could even suggest using "**elif**" as an alternative here.

How other parsers handle an else with condition. *CPython* only reports a "*syntax error*" after the "**else**", which does nothing to help the student. *Jython* offers some better error handling but

in this case it might lead a student into a completely wrong direction.

While the syntax `else x > 0:` is clearly wrong in Python, `else: x > 0` is actually legal. Upon encountering the else with condition, *Jython* hence writes that the colon is expected after the "`else`". A student who would write the following program is therefore implicitly asked to move the colon in line 3.

```
if x > 0:
    print "Positive"
else x < 0:
    print "Negative"
```

Jython's actual error messages reads: `mismatched input 'x' expecting COLON`.

A further error indicating that the print-statement in line 4 seems to be wrong could eventually lead to a "corrected" version like the following:

```
if x < 0:
    print "Negative"
else: x > 0
print "Positive"
```

Even though this program is syntactically correct Python code, it clearly does not express the intention of the programmer. It is therefore well worth to have the parser be extra vigilant and watch out for common patterns of students' misconceptions.

The data (programs and error messages) we have collected includes some instances where, indeed, students struggled with the correct placement of colons (cf. Chapter 8).

7.6.2 Misspelled Keywords

Initial keywords. Most statement types in Python can be identified by their initial or leading keyword. The parser can thus unambiguously determine the structure of the statement that follows an initial keyword. The keyword "`def`", for instance, is enough to inform the parser to expect the pattern: ⟨*def*⟩⟨*name*⟩⟨*(*⟩⟨*arg-list*⟩⟨*)*⟩⟨*:*⟩⟨*suite*⟩. However, if this initial keyword is misspelled, it can be difficult for the parser to figure out what statement or structure to expect.

Hence, how can the parser actually discern a misspelled keyword? What happens with the parsing process if the initial keyword is replaced by a ⟨*name*⟩-token?

As the underlying LL(1)-parser, which we have used as a basis, heavily relies on the initial keyword to select the statement, a misspelled keyword makes it impossible to continue correct parsing. In fact, a misspelled keyword becomes a ⟨*name*⟩-token, so the parser expects either an expression or an assignment and will treat the input accordingly (there are some exceptions of where the misspelled keyword transforms into another keyword, e. g., `def` and `del`).

First, in order for a ⟨*name*⟩-token to be accepted as a misspelled keyword, either the Damerau-Levenshtein ([10]) distance between the name and the keyword must be at most 1, or the name and keyword must differ only in case.

Second, the parser must have a clear indication that the statement in question is neither an expression nor an assignment. One such indication might be a colon at the end of the line and a suite attached to the line-node. Another indication is that the parsing process will not consume all tokens on the line.

Looking at the follow sets of both ⟨*name*⟩ and possible initial keywords (such as ⟨*if*⟩, ⟨*def*⟩, etc.), we find:

$$\textsc{Follow}(\langle \textit{name} \rangle) \cap \textsc{Follow}(\langle \textit{keyword} \rangle) = \{+, -, (, [\}$$

This means that the parsing process will only continue after the initial ⟨*name*⟩-token if any of the above four tokens is present. In the end, this leaves very few cases where the parser is not able to detect a misspelled keyword.

For instance, a misspelled "`print`" in `print(1, 2)` would go unnoticed – at least until the runtime system throws an exception because of an unrecognized name.

But most importantly, the parser can discern misspelled initial keywords quite reliably whenever the keyword determines the structure to follow.

Other keywords. This leaves us with keywords that appear inside statements and expressions, but not in an initial position. These include:

$$\langle \textit{and} \rangle, \langle \textit{or} \rangle, \langle \textit{in} \rangle, \langle \textit{as} \rangle, \langle \textit{if} \rangle, \langle \textit{else} \rangle, \langle \textit{for} \rangle$$

Once again, we assume that such a keyword is misspelled and the lexer produced a ⟨*name*⟩-token instead. How should the parser now discern any of these keywords?

All these keywords must directly follow an expression (or a subset of it, such as the ⟨*as*⟩ following a name). We therefore look at the follow set of expressions and find:

$$\langle \textit{name} \rangle \notin \textsc{Follow}(\langle \textit{expr} \rangle)$$

Hence, the parser can quite safely test for misspelled keywords in cases whenever an unexpected name occurs. However, an unexpected name must not necessarily be a misspelled keyword (it could also be due to a forgotten operator or comma, for instance). To decide that a name is indeed a misspelled keyword, the parser must also take the context into consideration and look at further tokens on the line. For instance, ⟨*if*⟩ must be followed by an expression and a subsequent ⟨*else*⟩ (note that an ⟨*if*⟩ which is not in initial position is part of the *if-expression*, not the *if-statement*).

Ambiguous cases. There are still several ambiguous cases. In particular, the collected data (cf. Chapter 8) shows that the improper statement `def square()` occurs in two situations. In some of the cases, the students seem to have started giving a definition for the function, but did not finish. In other cases, the function was already present in the program, and the statement was obviously intended to call the function.

7.7 Static Analysis

The static analyzer works on the AST and serves two purposes: it creates an extensive symbol table to assist the parser in case of syntax errors, and it checks the program for additional errors not caught by the parser so far.

In practice, the static analysis turned out to also work well as a system for auto completion and the display of documentation for any given function.

The goal of the static analyzer is twofold. First, it extends the error checking done in the lexer and parser so as to capture errors beyond mere syntax errors. Second, it recreates an image of all the objects, variables and fields used in the program. Ideally, it can then give basic information such as type for any given name at a specific position. This is used in the parser as described below.

Using type information in the parser. A frequent error of novice programmers is the omission of parentheses after a function's name to properly form a call. Just writing a name as a statement is legal in Python, but in the case of students it is safe to consider it an error.

The parser is capable of detecting such lonely names as statement without side effects. And it will properly report that the given statement is useless. In classroom, however, we observed that students did not understand what exactly was wrong with the code. The parser actually needs to report that the name must be completed with parentheses to form a valid call. Yet, such a message would be wrong unless the name really is a function. The parser hence must check the name's type and see if it indeed is callable.

Another error we want to capture concerns the incorrectly applied `and`-operator. Some students write `if a and b > 0`, meaning to say that both a and b must be positive. On one hand, the parser should correct this and report that a and b must both be tested individually. On the other hand, the above test makes complete sense if a is a Boolean value. Hence, before reporting the error, the parser must rule out that a could be a Boolean value.

These examples show why the parser indeed requires a static analyzer that can provide type information for any given name.

7.7.1 Type System

Building a static analyzer for Python is not an easy task. Not only do variables not have a specific type, even the scope and lifetime of a variable can be difficult to assess.

For this reason, the static analyzer pursues a pragmatic approach. Any variable is considered alive in its entire scope (even reaching "backwards"). Delete-statements are ignored. For any assignment to a variable the variable's type is the union of the current type with the new type. Hence, if, e. g., the two statements "`x = 1`" and "`x = 0.5`" both occur referring to the same

variable x, x has type "numeric", as the union of "integer" and "float", for the entire program code.

Objects: an example. Python is completely object-oriented in that each value is represented by an object, including a "`None`"-object representing the *null*-value (there are no "primitive" types as in Java that are not objects). Each object has a dictionary associating names with their respective values (in most cases the values are themselves just references to other objects).

Let us consider an example. In the following small code sample we define a class *Duck* with a constructor `__init__` and a method `eat`. Inside the constructor, we add the field `food` to the new instance. Further below an instance `duck` is created and eats "123 food".

```
class Duck:
    def __init__(self):
        self.food = 0
    def eat(self, amount):
        self.food += amount

duck = Duck()
duck.eat(123)
print duck.food
```

A simplified version of the objects involved here are depicted in Fig. 7.5 (for instance, the "integer"-class actually has many more methods not shown here).

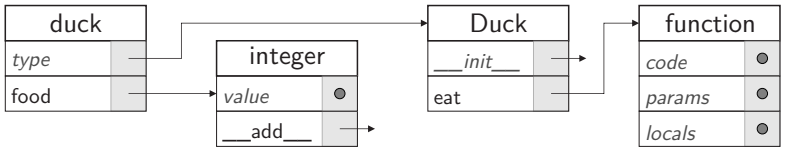


Figure 7.5: In Python, all values are actually objects, each with its own dictionary. While built-in objects have inaccessible private fields such as the *value* of an integer-object (represented here by a dot), most fields can be read and even written at will. When needed, new fields are automatically added to an object.

The statement `duck.eat(123)` prompts Python to find a field "`eat`" in the `duck`-object. As the `duck`-object itself has no such field, its base class `Duck` is searched where the field "`eat`" is indeed found. At that point, Python checks if the object referenced by this field is actually a function and if so calls that function with the `eat`-object as its first and the integer `123` as its second argument.

Python allows us to directly manipulate most of the fields of an object. We could, for instance, move the method from the class to the instance and rename it to `feed`.

```
duck.feed = duck.eat
del Duck.eat
```

The first statement copies a reference to the method/function and the second statement removes the reference from the `Duck`-class (it does not delete the function object itself, though).

A static picture of dynamic objects. This highly dynamic nature of Python's objects makes it difficult to create an accurate representation of the objects, its fields and values at any one time (without actually running the program, that is). Whether, in the above example, the fields `eat` or `feed` are available in the object `duck` depends on whether the additional lines with moving the method from the class to the instance have been executed or not. And since the original method `eat` has been deleted from the class itself, no other instance of type `Duck` would have such a method.

These ideas are not just academic and theoretical in nature. Some modules such as the official turtle-modules make indeed use of the dynamic nature and reflection. In order to make all methods of an object available as ordinary functions, they use code similar to the following (in reality the code is more refined and takes parameters into consideration as well).

```
turtle = Turtle()
for name in methods(Turtle):
    exec "def %s(x): turtle.%s(x)".format(name, name)
```

Instead of `turtle.left(90)` one can then just write `left(90)`. The function will automatically execute the method on the global *turtle*-object.

So, if we want to have an accurate picture of all names available in a program, we would have to actually execute the program and the loaded modules. This, however, is not an option for several reasons. Besides unwanted side effects stemming from executing imported modules, we clearly cannot execute the code we are in the process of parsing. This leaves us with the slightly unsatisfactory state that, without an enormous effort to consider all possible dynamic cases, the parser cannot be sure if a certain name does exist at any one location in the source code. In essence: the static picture we draw of the objects and types involved can only be approximate.

7.7.2 Discussion

From the perspective of the parser, the lifetimes of variables and functions are a complex issue. Reasoning about the existence, non-existence or the type of any name in the program is, in general, not possible. The parser therefore assumes that any object is alive from the moment of its definition until the end of its containing scope. At least in the case of novice programmers' program this is certainly a reasonable assumption.

Providing information about an error. Two examples of errors where the parser uses static analysis to establish the existence of a function are (a) `set color()` and (b) `forward 90`.

In the first case, `set color()`, the parser must decide whether the two names "`set`" and "`color`" could be combined to one identifier. For that, the parser needs to know if there is a function called "`setColor`", for instance (actually, the parser tries several combinations, including also, e. g., `set_color`).

In the second case, `forward 90`, if "`forward`" is an actual function, the parser proposes to put the argument into parentheses to yield `forward(90)`.

Otherwise, i. e., if the static analyzer cannot find a function with the given name, the parser assumes in both cases, that there is a comma or assignment missing.

Name- and type-errors. The above discussion should also make it clear why two of the most frequent and typical errors (cf. Section 6.2) are not syntax errors in Python but rather runtime errors: "*name not found*" and "*wrong number of arguments*". Both depend on the current state of the program.

Chapter 8

Experimental Results from Parsing Python Programs

8.1 Introduction

This chapter presents and discusses data we have collected over the course of two months. Users of the environment *TigerJython* had the option to send us their programs and error messages in an anonymized form. Judging from oral feedback, we can assume that at least three high schools asked their students if they would be willing to participate, and enable the sending of programs and error messages. Because of the anonymity of the submissions, we can not assess how many students (or users, respectively) actually participated.

The main question to answer in the course of this chapter is: do the produced error messages really fit the problems encountered in the students' programs? How close does the parser come to giving specific and possibly helpful feedback?

8.1.1 Theses

In the course of this chapter, we are going to provide support for one of our theses:

Thesis 3. The parser can correctly identify and report at least 75 % of syntax errors that are made by high school novice programmers in Python.

8.1.2 Organization

Section 8.3 presents the data collected as part of this study. In addition, it discusses several special cases, including those that gave rise to specific improvements.

Section 8.4 discusses to what extent the produced error messages might be helpful to the students. In particular, we argue that assessing the effectiveness of error messages is a complex task, and beyond the scope of this dissertation.

Finally, in Section 8.5, we conclude that our parser does meet the requirements to fulfill Thesis 3.

8.2 Methodology

To assess the quality of our work, we started to collect programs and their error messages directly from the students using it. Our educational Python environment *TigerJython* offers the choice to participate in this study by sending the respective data anonymously to our server in Switzerland.

Some of the received data turned out to be invalid because of transmission problems – sometimes we did not receive the entire programs, or even just error messages without any program to assign it to. For our study, we only considered error messages with an underlying program and a valid error position within that program.

Since our interest is with the parser and its performance, we also filtered out all runtime error messages. They were produced directly by the Python interpreter itself and not by our parser. In addition, we removed all error messages that were produced by Python's parser. Those that were in our data set seem to have come from participants who disabled the extended parser discussed in this thesis. Accordingly, we could not gain any information about our parser from them.

For each of the remaining error messages, we then looked at the message, compared it with the associated program code and decided if the message (a) accurately described the problem, (b) was technically correct but not necessarily helpful, or (c) completely wrong.

The data was collected in two tiers, each with a window of about a month. After the first tier, we looked at the data and made small improvements to the parser so as to better handle cases that were not entirely correct. We then published the updated version of the parser (as part of the *TigerJython*-environment), and collected again data for more or less the same period of time.

Anonymization. Before any program is sent to our servers, all string literals are blacked out using the letter "x". That is, the length of the string literals is preserved but all letters are replaced by "x". Numbers just retain their first and last digit with all other digits set to "0", and comments are replaced entirely by the comment character "#". It is important for our purposes to keep the number of characters in the programs so that we can correctly assign error messages to the correct locations within the program.

8.3 Collected Data

8.3.1 First Tier

We have collected data from at least three different high schools (based on oral feedback from the respective teachers). Within one month (from August 26 to September 27), we collected 6955 individual submissions with each submission being a program or an error. We counted a total of 916 error messages in these submissions. 235 submissions could be attributed to the "*name not found*" error.

Some of the submissions were corrupted, and did not contain all necessary data for the analysis. In some cases an error message was produced repeatedly for the same program, i. e., without any change to the underlying program. We counted such repeated submissions only as one instance.

We used a program to automatically filter these submissions and only show those with fully valid error messages and fitting programs. This resulted in 387 valid error messages that were produced by our parser.

After manually going through the 387 valid error messages, we found 5 cases that were completely incorrect and 71 cases where the error messages were technically correct, but did probably not address the actual problem (hence, we deemed abuot 80 % of the error messages to fit the students' problems). The following paragraphs discuss some of these cases.

Technically correct but not helpful. Technically correct but not helpful messages included the cases of generic error messages such as "*no viable alternative*". Such an error message is clearly no more specific than what standard Python compilers already offer. Further examples of this category are given below (we took the liberty of slightly simplifying and shortening some of the examples for better clarity).

- **You cannot assign something to 'var aa'.**

  ```
  var aa = input(a)
  ```

  The student obviously brings prior programming knowledge from another language such as JavaScript. The parser could inform the student that Python does not require the "`var`"-keyword.

  Interestingly, after the error message the student tried the "`var aa`" without the assignment, which led to the error message "*missing assignment.*". This message, of course, refers to the idea of writing "`var = aa`", but the student probably understood it differently.

- **There is a name required here.**

  ```
  myList = ['abc'.'abc', 'abc']
  ```

The error message is reported because the dot after the first string literal indicates that the name of a field should follow. In this case, however, the actual error is rather a typing error where the dot should have been a comma instead.

- **A double equal sign '==' is required here.**

```
while inputValue = input(''):
    myList.append(inputValue)
```

Again, the student seems to come with prior programming knowledge from another language. It was obviously not the intention of the student to compare the variable "`inputValue`" with the new value entered by the user. Yet, the error message might be close enough so as to inform the programmer that Python does not support assignments in tests.

- **':' required but '(' found.**

```
def square(s)():
    ...
```

Instead of telling the student that a colon is required the parser should actually see the colon and report that the second set of parentheses is superfluous and must be removed.

- **There is/are extra token(s): 'h:'.**

```
print hours "h:", minutes, "m"
```

The actual error here is the missing comma and not the extra string literal. This is actually one of the hardest situations to correctly identify because the correct alternative strongly depends on the context and intentions of the programmer. In principle it would also be possible to insert an operator between the two tokens "`hours`" and "`"h:"`".

- **There is a comma missing.**

```
setPenColor(sky blue)
```

The error here is, of course, not a missing comma but rather missing quotation marks. "*sky blue*" should be a string literal in this case.

Not correctly identified. The five cases of not correctly identified problems can be represented by three examples. The first example with the invalid **`for`**-statement was submitted in total three times with slightly different versions.

- **Colon required but '==' found.**

```
for v==1:
    print 1
```

This **`for`**-statement is not missing the colon but rather the keyword "**`in`**" after the variable.

- **There is/are extra token(s): '{'.**

```
n = 4
switch(n) {
  case 0:
    printf("Abc");
    break;
  case 1:
    ...
}
```

This is another instance of code from another programming language directly copied into the Python environment. The parser could detect this pattern and report that Python does not have a switch-statement.

- **Unexpected symbol: 'or'.**

```
or code in inputCodes:
    output.append(f(code))
```

During the construction of the parser we assumed that a misspelled keyword would result in a ⟨*name*⟩-token. In this case, however, the misspelled ⟨*for*⟩-token became another keyword ⟨*or*⟩ and hence was not discovered by the parser. Even though the data shows that the students quickly understood and fixed the problem, the parser should be able to correctly detect this error.

8.3.2 Second Tier

Our second window for collecting data was open slightly more than a month (from September 30 to November 10), due to school holidays in October. We collected a total of 7639 submissions, including 1611 error messages. 605 of these error messages were runtime errors from the Python interpreter. We could attribute 406 runtime errors to the "*name not found*" error.

After an automatic filtering of invalid and repeated error messages as before, we were left with 475 submissions (cf. Section 8.3.1). We found that the provided error messages matched the problem in about $80\,\%$, or 396 of these cases. A detailed account of all 475 cases can be found below.

Accurately describes the problem	396	$83.2\,\%$
Technically correct but not helpful	41	$8.6\,\%$
Not correctly identified	39	$8.2\,\%$
Total	475	$100.0\,\%$

While classifying the submissions we noted that not all students participating in the study seemed to have used the latest version of the parser. In fact, one school confirmed that it would not update any computer program in mid-semester. Accordingly, we cannot reliably assess the improvements made to the parser.

For instance, in 17 cases the submission recorded the error message "*This statement is useless: it has no effect*" when, in fact, a name was missing parentheses. The improved version of the parser, however, would report "*To call a function you must add parentheses even when they are empty*".

Detailed account of the analyzed submission. For each reported error message, we give the number of correctly identified error instances, and, where appropriate, further details about the recorded instances. In addition, we also describe cases of where the reported error would not match the true problem found in the program.

All submissions were already anonymized as far as possible. In order to keep all instances completely anonymous, we decided to limit the examples to single lines. Furthermore, we changed all identifiers to generic names such as "square".

- **A colon ':' is required here.**

  37 correctly identified. In 16 instances the colon was missing after an otherwise correct `repeat`-statement, in 21 instances after an otherwise correct `def`-statement.

  In one instance the reported error was technically correct, but it did probably not address the real problem: `repeat left(90)`.

- **There is/are extra token(s): 'X'.**

  22 correctly identified, 6 instances were clearly wrong, 23 instances were technically correctly identified, but did not address the problem.

  In 10 instances the parser correctly reported an extra `def` as in, e. g., `def square(100)`. In 9 instances the student had added an extra colon after an otherwise legal function call, e. g., `forward(10):` or `setPenColor:("red")`. Another 3 instances were miscellaneous extra symbols as in, e. g., `forward(=s)`.

  There were 23 instances of the parser incorrectly reporting an extra `def` for inputs like, e. g., `def square()`. In constrast to the 10 correctly identified instances above, the students seems not to have meant to call the function, but rather define it.

  In three instances, the parser reported an extra colon for `repeat8:`, which clearly is not correct. It should have reported a missing space between the `repeat` and the `8`. Also incorrect is the same error for the input `defsquare():`. In a further incorrect instance, the parser reported the extra `import`-keyword for the input `fro, gturtle import *`.

  In a last case, the parser reported an extra `from` for the input `1 from gturtle import *`. In this case, it is obviously the "1" at the beginning of the line that is wrong, and stems from line numbers.

- **Invalid definition of a function.**

  8 correctly identified. In 7 instances, the `def`-statement was incomplete, i. e., `def name`. In one instance, the syntax was incorrect: `def square ":"( )`.

- **This is an invalid name: 'X'.**

  29 correctly identified. All instances were function definitions with an incorrect name. In 22 cases, the name contained an extra space, in one case it started with a digit and in 6 cases it contained an invalid character such as a German umlaut.

- **There is a body or indentation missing.**

  82 correctly identified. 43 instances came from a `repeat`-statement without proper body, the remaining 39 instances from a function definition without proper body.

- **There is a comma missing.**

  Of the 32 instances, none seemed to completely catch the error. However, 22 instances should have resulted in better error messages.

  In 17 instances, the parser should have reported an extra space as in, e. g., `make Turtle()`. In four instances, several statements had been written to a single line, requiring a semicolon (and not a comma as indicated), e. g., `penUp() forward(50)`. In one instance, there were missing parentheses, i. e., `forward 90`.

  The remaining ten instances were of different natures, including, e. g., `setPenColor(light blue)`, `rep 4:`, `problem 2:`, etc.

- **There is a name required here.**

  10 correctly identified. In eight cases, there was a number given as parameter, e. g., `def name(123):`. In one case, there was the name of the function missing, and in one case the name was given as a string literal: `def 'xxx':`.

  4 instances were not correctly identified. In these cases the colon was in the wrong place, e. g., `def: square`.

- **Parameter(s) required but ':' found.**

  14 correctly identified, e. g., `def name:`. Two instances were not correct: `def square:()` and `def square,rotated():`.

- **X required but Y found.**

  1 correctly identified, 1 incorrect error message.

  The first instance was `def square();` with the message, that a colon would be required instead of the semicolon. In the second instance with `from, gturtle import *` the parser should have reported an extra token instead.

- **The result of this expression is never used.**

  All four instances were incorrectly identified. In three cases, the student had put a line number at the beginning of the line, e. g., `2 makeTurtle()`. This led the parser to insert an operator and then conclude that the result of that calculation would never be used. The final instance was a simple typing error: `square/()`.

- **This statement is useless: it has no effect.**

  19 correctly identified, 6 instances were technically correct, but did not address the source of the error.

  In 17 cases, the student had written a function's name without parentheses, and in two cases a calculation like, e. g., `12/4`. The six final instances were of the form `repeat: 4` with swapped tokens.

- **To call a function you must add parentheses even when they are empty.**

  51 correctly identified, e. g., `penUp`.

- **This import statement is incomplete.**

  12 correctly identified, in each case with a missing star at the end: `from gturtle import`.

- **There is a missing left bracket or parenthesis: '('.**

  3 correctly identified, e. g., `left90)`.

- **There is an operator or comma missing.**

  4 correctly identified, e. g., `forward(2xside)`, `dot(2d)`.

- **There is a closing bracket or parenthesis missing: ')'.**

  28 correctly identified. In one additional case there was an extra opening parenthesis instead: `(forward(10)`.

- **There is a space missing.**

  3 correctly identified, all three with the `repeat`-statement: `repeat4:`

- **Missing 'def'.**

  3 correctly identified of the form `name():` with a subsequent body.

- **Misspelled keyword 'X' instead of 'Y'.**

  3 correctly identified: `gdef`, `ddef` instead of `def`, and `repea5t` instead of `repeat`.

- **These tokens seem to be swapped: ':' and '2'.**

  1 correctly identified: `repeat: 2`

- **This string is unterminated.**

  6 correctly identified.

- **Unexpected end of line or input.**

  1 correctly identified.

- **A 'function'-definition cannot be inside a loop.**

  1 correctly identified.

- **This function sometimes returns a value and sometimes does not.**

  1 correctly identified.

- **This is an invalid input character: 'X'.**

  2 correctly identified, in both cases due to a German umlaut.

- **A double equal sign '==' is required here.**

  1 correctly identified: `elif i = 2:`

- **There is an extra indentation.**

  40 correctly identified.

- **The indentation is inconsistent.**

  12 correctly identified.

- **There is an extra opening bracket or parenthesis: '('.**

  1 correctly identified: `left((90)`

- **There is an extra space.**

  1 correctly identified: `draw Square()`

8.3.3 Student's Misunderstandings

In a few submissions, we were able to track the student's response to the error message provided. While a direct comparison was often difficult due to too many changes made to the program code, we found a few noteworthy examples.

The three examples below show clearly that students need to already have a basic understanding of Python's syntax in order to fix the program code. Otherwise, the error messages can be interpreted incorrectly.

Some of the examples show the original code on the left, and the student's resubmission after the error messages has been shown on the right. As before, we changed some of the names and values in order to keep complete anonymity.

"There is a body or indentation missing." Students sometimes forget to properly indent code that belongs into the body of a loop or function. Most of these errors seem to be due to incomplete modifications of existing code. For instance, when inserting the line "`repeat 4:`" above existing statements in order to repeat them. However, in some cases the corrections made by some students suggest that those students do not have a proper understanding of the indentation principle. One student reacted to the error message "*There is a body or indentation missing*" by indenting not only the loop's body but also the "`repeat`"-statement along with it.

```
repeat 2:
square(30)
square(50)
square(70)
```

```
    repeat 2:
    square(30)
    square(50)
    square(70)
```

Another student indented the "`repeat`" instead of the body as shown below.

```
def square(side):
    repeat 4:
    forward(side)
    right(90)
```

```
def square(side):
        repeat 4:
    forward(side)
    right(90)
```

One student seems to not have understood how to actually correct the problem of missing indentation of a loop's body. The data shows a total of six iterations for the following program with no actual progress.

```
def square():
    repeat 4:
    forward(100)
    right(90)
```

At first, the student inserted parentheses between the "`4`" and the colon so that the line read "`repeat 4():`". As this did not work, he then removed the parentheses and wrote "`repeat4:`" without a space before putting the parentheses back.

Yet another student corrected a "`repeat 4:`"-statement without any body after four different attempts to "`repeat 4:4`" and back to the original version before giving up.

"To call a function you must add parentheses even when they are empty." In order to actually call a function the function's name must be followed by parentheses. Particularly in the case of functions without parameters and hence empty parentheses, students tend to leave them out completely and must be reminded of the necessity of parentheses by the error message. One student, however, misunderstood the hint and put the parentheses around the function's name as shown below.

```
repeat 8:
      drawStar
```

```
repeat 8:
      (drawStar)
```

A matter of colons. The program shown below suggests that the student struggles with the concept of colons. There are three extra colons as part of function calls and a missing colon in the function's definition.

```
from gturtle import *
makeTurtle ():

def square
    repeat 4:
        forward(100)
        right(90)

setPenColor: ("red")
square: ()
```

At first Python indicates that there is an extra token `':'` in line 2. However, rather unexpectedly, the student removes the parentheses in line 2 instead of the colon so that it reads `makeTurtle:`. As Python displays the same error message again, the student then removes the colon, too. Python therefore prompts him to put parentheses after the `makeTurtle` in order to actually call the function.

After correction of that error, Python points out that the definition of the function in line 4 is invalid. The student reacts by adding parentheses after the function's name. When Python indicates a missing colon in line 4 the student puts the colon not at the end of the line but rather between the function's name and the parentheses, resulting in `def square:()`. It takes two additional steps until the syntax of the function is correct.

The extra colons in the last two lines, however, are each corrected in a single step, indicating that by now the student understood that he must indeed just remove these colons.

Invalid definition of a function. In this example, Python asked the student to provide parameters for the function.

```
def jump:
    (100)
```

However, instead of adding parentheses the student put the function's name in single quotation marks, i. e. `def 'jump':`. After Python then indicated that a name was required the student added some parameter inside the string, leading to `def 'jump(distance)':`. As this still did not turn out to be correct, the final submission shows the `def`-keyword in quotation marks instead: `'def'jump(distance):`.

8.3.4 Extra Whitespace

Among the more frequent student errors were extra spaces, particularly in identifiers that were made up of multiple words, such as, e. g., "`setPenColor`". Students would then write, for instance, "`set pen color`", resulting in the syntax error of two consecutive identifiers.

Identifying the cause of two consecutive identifiers is a difficult task for the parser, because there are several actual errors that all lead to the same syntax error. Consider, for example `set color`, leading to the following list of actual causes and intended statements.

- extra space, i. e., `setcolor`,
- missing assignment, i. e., `set = color`,
- missing comma, i. e., `set, color`,
- missing operator, i. e., `set + color`.

Identifying the error's cause. At first, the parser always reported a missing assignment – which is only rarely correct. Hence, an " improved" version, that was also used for the second tier of data collection, performed several checks in order to find the true cause for the error.

For the two juxtaposed identifiers "`set color`", the parser would first look if either "`setcolor`" or "`set_color`" is a known identifier. At this point, the lexer's symbol table comes into play (cf. Section 7.4.1). The parser also performs static analysis so as to take imported names into consideration. If either concatenation of "`set`" and "`color`" is found to be a properly defined name, or to be used in other places of the program, the parser decides that there is indeed an extra space.

Why it did not work. In the collected programs, we found that the error of an "extra space" usually occurs in connection with a change of case. Students have not only inserted an extra space, but also changed the capital letter of the second word to a lower case letter. For instance, "`set color`" should not be corrected to "`setcolor`", but rather to "`setColor`" with capital "C".

Secondly, some of the names had more than one space inserted, since they consisted of more than two names. For example, "`set pen color`" instead of "`setPenColor`".

Finally, students were often more or less consistent in splitting names. This resulted in several incorrect identifiers occurring in the program. The parser therefore assumed that the given name parts were intended as full identifiers.

In short, the requirement that each syntax error occurs in isolation (Section 7.2.1) does not hold.

8.4 Do Error Messages Help the Student?

Particularly in the context of education, a most relevant question is: *do the improved error messages actually help the novice programmer?* What is the effect of the error messages on the students?

We found that measuring the effectiveness of (enhanced) error messages is a hard problem. Moreover, obtaining reliable and generally valid results is clearly beyond the scope of this dissertation.

8.4.1 Do Error Messages Help in Learning – A Survey

We conducted a survey to assess what students found most helpful for their learning in programming. The survey's questions were in part inspired by a study of Lathinen et al. [33] on the difficulties of novice programmers. While many questions of the study by Lathinen et al. did not apply to our situation, we were able to reproduce one finding of that study: students rated *example programs* as the most helpful materials for learning in both surveys (Lahtinen et al. found an average of 4.19 with standard deviation 0.86).

Methodology. The students completing our survey were asked to rate questions on a scale from 1 (not helpful, do not agree) to 5 (very helpful, strongly agree).

In total, 82 students from three different high schools as well as a university of teacher education answered our survey. All students had about seven month of training in programming with Python. In addition, the surveys were conducted in regions where German is the native language. However, not all students provided answers to all questions.

Results. The results from the survey are collected in the table below. However, we only included questions with some relevance to this thesis.

For each question, we give the number of participating students, the average score of the answers (from 1 to 5) and the standard deviation.

	N	Avg	Std
Of how much help are the following items for your learning?			
(1) Explanations in the course materials	82	4.04	0.83
(2) Studying examples in the course materials	82	4.15	0.93
(3) The error messages shown in the environment	81	3.30	1.20
(4) Running a program step by step with the debugger	52	3.00	1.16
How strongly do you agree to the following statements?			
(5) The German translations of error messages are helpful	53	4.04	1.09
(6) I do not read the error messages	54	1.90	0.94

Discussion. With a score of only 3.3 (slightly above average), the error messages were not considered particularly helpful for learning (question 3). However, the standard deviation is relatively high, indicating the students' opinions on this issue varied.

Of the 54 students who answered question 6 about whether they read the error messages at all, 11 students had a score of three or more, and two students strongly agreed. Hence, in general, students answered that they would read the provided error messages. This result contradicts somewhat our experience from classroom. While teaching, we felt that many students would just discard the error messages without paying any closer attention to them. Even when considering that our own experience is highly anecdotal, the answers to this quesion might not be entirely reliable. In addition, due to the negative phrasing of question 6, we have to assume that some students put their marks in the wrong field.

We also compared the answers to questions 3 and 6, but we could not find any correlation. Even among the two students answering that they would not read the error messages, one found them very helpful (5) while the other rated them as not helpful at all (1).

Note that the students answers suggest a high appreciation of the error messages' translation into their native language. Given the high score of question 5, we could hypothesize that students rate error messages higher if they are given in an understandable language, whereas the accuracy might be of less concern.

Conclusion. Based on these numbers, the improved error messages did not stand out as particularly helpful in learning to program. The most appreciated detail about error messages seems to be a translation into the students' native language.

8.4.2 Related Work

The importance of a compiler's error messages for the student have long been recognized. For instance, du Boulay et al. see error messages as an important source of information about the notional machine that runs the program. They state that "*error messages [. . .] form an important window into the machine*" [14]. However, the actual messages produced by widespread compilers seem of little help to the novice. As, e. g., Nienaltowski et al. note: "*Novices find it difficult to understand and use compiler error messages*" [43].

Accordingly, various projects and studies have sought to improve a compiler's error messages [5, 11, 24, 43]. Becker [5] and Denny et al. [11] both provide good overviews of other related work, but also conclude that few of these projects provide an empirical evaluation to its effectiveness. For instance, Becker notes that "*many of the studies discussed [. . .] focus on addressing the problem with compiler error messages, but lack empiricism in determining if they make any difference, particularly to novices*" [5].

Most of the studies use Java as the underlying programming language, none uses Python (cf. Section 6.1).

Improving error messages. While the different studies have tried to improve the error messages in various aspects, no consensus on the effectiveness of these improvements has been reached.

Denny et al. [11] enhanced Java's syntax errors in two ways. First, they built a recognizer for common error patterns so as to accurately identify 53 different types of syntax errors. They then enhanced each error message to not only include a short message, but also a more detailed description of the problem, as well as an example.

In order to assess the success of their approach, Denny et al. measured how often each student attempted to compile a program before it would actually compile and run. However, they conclude that "*Although we anticipated that the enhanced error messages would help students to identify and correct errors, analysis of the data shows no significant (or practical) effect.*" [11]

A second study by Nienaltowski et al. [43] investigated compiler error messages for Eiffel and Java, respectively. Error messages were presented to the student either in a short, concise form, visually by highlighting the error inside the editor, or in a "long form" with a more detailed explanation.

The actual study of Nienaltowski et al. was based on a questionaire, and measured the number of correct answers, as well as response times for some of the answers. The study also concludes that the additional information provided in the longer error messages "*did not aid message comprehension, or help identify the error faster or better*" [43].

In contrast to the first two studies, Becker [5] found that enhancing compiler error messages is helpful. The tool used in Becker's study recognized a total of 30 common Java errors, and generated customized error messages for each of these errors. The assessment was then based on the number of iterations per error, as well as per student. The study concluded that in the case of eight error types, the enhanced messages resulted in a significant reduction of the number of errors.

Measuring the effectiveness. The studies discussed above use slightly different methods to assess the effectiveness of enhanced error messages. While Denny et al. [11] counted the number of attempts to run a program before it would actually compile, Becker [5] counted the numbers of individual errors. Finally, counting the number of correct answers as in the study of Nienaltowski et al. [43] can only be used in the context of a dedicated survey or questionaire.

However, as noted by Marceau et al.: "*There is no single metric for 'effectiveness' of an error message*" [39]. The study by Marceau et al. measures the effectiveness of an error message by the student's response to it: "*does the student make a reasonable edit [. . .] in response to the error message?*" [39]. They also contrast their method to comparing the students' grades.

In the light of the different studies presented here, we can conclude that an accurate measurement of the effectiveness of enhanced error messages is a hard problem.

8.4.3 Discussion

When assessing the effectiveness of error messages, we face different problems. First, we need to specify what we mean by "effectiveness". Is an effective error message one that supports the novice's learning as good as possible, or rather one that lets the novice correct his or her program as quickly as possible? Second, we must find an appropriate measurement for the quality we are interested in. For instance, how do we decide whether an error message has successfully helped the programmer to overcome a problem? Third, we have to take the vast inhomogenity of the student population into consideration. Which kind of students or novice programmer do we want to address and take into our study?

What is effectiveness. Enhancing error messages is frequently understood as providing the student with additional information and explanations. Tools like the one described by Denny et al. [11] do not only point out the error, e. g., "*missing parentheses*", but also give an extended explanation as of *why* parentheses would be needed in the case at hand.

Our own anecdotal experience from classroom, however, contradicts the effectiveness of such enhanced error messages. We observed that students rather quickly discarded the lengthy message – without ever reading it (fully). It seemed as if the students perceived the explanations as a distraction that had little to do with their own code. Even though we have no evidence, this assumption is supported by the study of Nienaltowski et al. [43] that found no support for longer error messages as more effective, even while stating that "*the assumption is that longer explanations and suggestions of error corrections improve novice's understanding of the problem and therefore their performance*" [43].

Instead of attempting to directly teach students the reason for the error at hand, error messages might need to enable a student to quickly fix, compile, and run the code. The learning process would then have to be considered in a larger context.

Finally, good error messages might not only instruct the student how to fix the program code, but also reveal some of the machine's inner working, as has been proposed by du Boulay et al. [14].

How to measure effectiveness. A valid statistical measurement requires a large number of sample data. This, in turn, implies an automated assessment of the data. Possible quantities that might be measured to that end could include the number errors and failed attempts to run a program, or the time between the display of an error message and the subsequent attempt to run

the program. A much coarser grained study might look at the students' grades, thereby gaining little insight into which error messages proved helpful.

One might also measure the number of error-related questions asked by students in a classroom. The implication being that students who are provided with good error messages would require less assistance from a teacher or tutor.

The data on error messages collected as part of this thesis included the time intervals between the display of an error message and a subsequent resubmission of the program. This would, in principle, allow us to measure how long it took students to correct errors as based on the error messages.

However, we found that the times as well as the changes made to the programs varied so strongly that no meaningful measurement was possible. The most simple syntax errors (e. g., the missing parenthesis in `forward40)`) were often quickly fixed, but it is doubtful how much these simple cases reveal about the effectivness of the error message. In many more complex cases, the resubmitted programs showed significant changes. Accordingly, it is difficult, and frequently a matter of interpretation whether the changes just solve the problem pointed out by the error message, are an abandonment of the original idea, or also include further extensions not related to the original error.

Moreover, Marceau et al. [39] state that the data collected on error messages does not support any conclusions about conceptual difficulties "*because the error* message *that a student sees is often not a direct indicator of the underlying* error" [39]. In addition, du Boulay states that "*Often the error message is not very explicit about what the novice has done wrong.*" [13]

Finally, we observed in classroom that students frequently interrupt their work on the computer, for instance in order to help each other, or discuss a colleague's solution. From this experience, we consider time measurements highly unreliable.

The diverse student population. Section 8.3.3 shows that some students exhibit a very poor understanding of the basic structure of Python programs. Some submissions even included pseudo-statements such as, e. g., `Problem 1:`, or what appears to be a literal copy from the textbook:

```
def name(parameters):
    here go the statements
    that are part of your
    function.
```

What message should an error message addressed to these students convey? Is it even possible to help foster an understanding of basic principles in error messages?

On the other side of the spectrum, we find submissions that seem to stem from students with prior exposure to programming in another programming language. The following switch-statement shows that the student already grasped the basic syntax of Python, and tried to translate a foreign structure to Python:

```
switch n:
  case 1:
    ...
  case 2:
    ...
```

How would, in such a case, a correction of the error look like? Can we, in less obvious cases, discern between an advanced student trying to apply previous knowledge, and a complete novice who just copied a piece of code from the internet?

8.4.4 Conclusion

Judging by the work that has already been done, as well as the data we have collected so far, assessing the effectiveness of error messages is clearly a difficult problem. The conceptual problems, such as what makes an error message effective, are supplemented by administrative challenges, including the subject for the study.

We therefore conclude that such an assessment on the effectiveness of the error messages produced by our parser is beyond the scope of this dissertation.

8.5 Discussion

Syntax errors are a class of strongly differing types of errors. Some errors can clearly be attributed to a simple or even trivial typing error. Other errors rather expose fundamental misconceptions of the student, and a lack of understanding of Python's syntax. In the first case, a short hint might usually suffice to help the student correct the program code. The second case, on the other hand, might require more elaborate intervention. However, good error reporting is even further complicated by the fact that the "true" errors of the students do not map one-to-one onto the errors seen in the faulty program.

Consider the example of the erroneous statement `def square()`. In the collected data, we found both instances where this error was due to a missing function body, and where the student's intention was rather to just call the function "`square`". Hence, the two error sources map to the same syntax error. Moreover, we might want to interpret the missing body as a mere oversight, just an incomplete program that is missing some parts. At the same time, the "`def`" in the call-statement might be due to a conceptual misconception.

Hence, writing a parser with good error recognition is not only an engineering task. The students' actual mistakes, syntax errors and problems must also be taken into consideration. To this end, we have collected the programs and error messages of novice programming students over two months.

8.5.1 Assessing the Results

The collected programs and errors indicate that the population of programming students is extremely diverse. While the produced error messages might be useful for some students, the same situation might be unintelligible to others.

Even measuring how good the reported error messages fit the actual problem is a subjective task, since we often have limited knowledge about the student's plans, intentions and understanding of Python's syntax. Moreover, the number of fitting messages decreases quickly if we take into account the submissions where the programs turned out to be prose text, for instance, and not Python code at all.

8.5.2 Thesis

Thesis 3. The parser can correctly identify and report at least 75 % of syntax errors that are made by high school novice programmers in Python.

Despite the difficulties associtated with syntax error, and the assessment of how "correct" the reported errors are, we have shown that, under reasonable assumptions, the parser can produce accurate error messages for a large set of common syntax errors. In both tiers of data collection, the parser correctly identified around 80 % of the syntax errors.

Chapter 9

Conclusions and Future Research

9.1 Studying the Errors of Students

Teaching students and observing their errors. Teaching programming is difficult. Our students do not only come with preknowledge from other fields, particularly mathematics, but also form new misconceptions based on the material they are presented with. As educators, we must help the students identify and correct their misconceptions about programming.

The primary tool for recognizing and identifying misconceptions is studying the students' errors and mistakes. The relationship, however, between the students' actual understanding (or lack thereof) and the observable behaviour and errors is complex and depends on many additional factors. In other words, a single error usually gives us little information about the students' thinking.

Theses 1 and 2: a model of variables. If we look at collections of errors, patterns start to emerge, and we can form a clearer picture of the students' misconceptions that led to the observed errors. We presented one such pattern, which indicates that some students apply a model of mathematical reasoning to programming, leading to incorrect assumptions about how programming actually works.

As part of this dissertation, we have not only given an interpretation of specific error patterns. We have also proposed a teaching method to address the problems, and provided some indication that our teaching method is indeed effective.

Thesis 3: recognizing syntax errors. As not all errors are due to an underlying misconception about fundamental principles, we might also want to provide the student with automated, yet intelligible feedback about his or her (syntax) errors. Such feedback comes directly from the programming environment in form of error messages. Precise and accurate feedback could help

guide the novice programmer towards implementing his or her algorithmic plans. Yet, such a guidance requires a preceding thorough study of error patterns.

In our dissertation, we have presented a collection of syntactical and semantical errors found in students' programs. Based on that collection, we then have implemented a parser, capable of identifying many of these errors. Subsequent comparison to further programs by our students showed that there is still a considerable percentage of instances, where the parser was not able to correctly identify the error. However, this is to be expected: as pointed out above, the relationship between students' misconceptions and the errors found in programs is too complex to automatically infer the underlying problem from a syntax error.

9.2 Future Research

Developing a correct mental model of the notional machine. A proper mental model of how a program is executed is essential in the field of programming. Developing such a correct mental model of the notional machine (i. e., the conceptual machine that actually executes the program) turns out to be a difficult and error-prone process.

In our thesis we have shown that some students base their mental models of the notional machine on mathematics. They thus attribute algebraic capabilities to the executing machine, resulting in incorrect programs and interpretations of program execution. Taking into account that our students are extensively trained in mathematics, we might assume that students with different backgrounds construct different mental models of the notional machine, still based on preknowledge from another field. As reported in various other studies (cf. Section 5.2), some students also base their model on, say, the use of language, thereby assuming, e. g., that a variable called "max" will automatically hold the maximum value of a given data set.

Further research will undoubtedly reveal further sources of misconceptions and inappropriate mental models. Such research will also study how common certain models and misconception actually are. Of special interest, of course, is the question of how we can teach our students so as to correct their misconceptions and foster them in developing the proper mental models.

Syntax errors in Python. With a Python environment that is capable of discerning different syntax errors, future research can start to investigate the frequency of syntax errors in Python, and identify the most common syntax errors. Educators could direct their students' attention to common problems and concentrate on what is really difficult.

Of even more interest is a proper evaluation of how helpful the parser's error messages actually are. As we have pointed out in Section 8.4, such research will be extensive and requires careful selection of study subjects, as well as the metrics to measure. Possible metrics include:

- The time to correct (syntactically) faulty programs: how long do students take to correct a

syntax error, how often do they resubmit a program until it is (syntactically) correct?

- The assistance students need from their teacher or tutor to correct a (syntactically) faulty program: how frequent and how extensive do students require additional explanation from an expert?
- The learning effect: how much do students know or learn about Python's syntax with the help of error messages?
- The students' self-perception and self-confidence: do they find the provided error messages helpful, and, in particular, do they feel supported by the error messages in the implementation of their plans and goals?

Errors and miconceptions. Studying the students' errors and mistakes remains an important tool to paint a picture of their understanding and misconceptions about programming. As we have pointed out above, individual errors usually tell us little about the students actual thinking. Hence, it would be most illuminating to investigate the sets of errors made by each individual student, and compare them to the students performance and ability to correct his or her mistakes based on the given feedback.

Syntax Errors

The following list includes all errors that can be reported by the parser as presented in this thesis. We give a very brief description of each error.

Not that the internal names, by which this list is sorted, have often been parametrized later on. For instance, AND_CONNECTS_CMP_NOT_VARS does not only apply to `and`, but also to `or` (which has been added later on).

The parser has two different modes to operate in. In "*strict mode*", the parser does not only recognize true syntax error according to the grammar, but also some additional problems. For instance, in strict mode, a simple name is not a valid statement. Errors only reported in strict mode, are marked with an asterisk `*`.

AND_CONNECTS_CMP_NOT_VARS *'X' connects comparisons not variables.* where `X` is either `and` or `or`.

This error can be explained as a "natural language misconception" where students write `if x and y > 0` instead of `if x > 0 and y > 0`. In order to detect the error, the parser looks for the pattern where `and` or `or` connect a single variable name with a full comparison. Before reporting the error, however, the parser checks if the name on the left (`x` in the example above) has type 'Boolean'.

See also: CANNOT_TEST_TUPLE.

ARG_AFTER_VARARGS *You cannot pass further arguments after an unpacking var/keyword-argument.*

When calling a function, you can unpack a sequence such that each element of the sequence is treated as a single argument. After this unpacking, however, no further arguments are allowed. For instance, `foo(*mylist, 123)` is illegal.

AS_NOT_ALLOWED_HERE *'as' is not allowed/supported here.*

In the future-import, i. e., `from __future__ import ...`, you cannot use an alias for the imported name because there are no names to be imported into the dictionary of locals.

ASSIGNMENT_TO_RIGHT *The target of an assignment must be on the left.*

Assignments are often a source of errors and misconceptions. In particular, the equal sign '=' suggests a symmetry, leading students to believe that both `x = 2` and `2 = x` are valid assignments.

When the parser detects that the target of an assignment is invalid, it checks whether the source is a single name. If the right-hand side of the assignment is indeed a name and hence a valid target for assignment, this error is reported. Otherwise, the parser reports an INVALID_ASSIGNMENT.

BREAK_OUTSIDE_LOOP *A 'X'-statement cannot occur outside a loop.* where `X` is either **`break`** or **`continue`**.

Standard error.

CALL_NEEDS_PARENTHESES* *To call a function you must add parentheses even when they are empty.*

When the parser detects a single name used as a statement, it checks if the given name is a function. If so, it reports that the function call needs parentheses instead of USELESS_STATEMENT.

There is one case, however, which the parser will accept: when the name occurs inside a **`try`**-block. It is a common pattern in Python to test if a given name has been defined or a module has already been imported.

CANNOT_APPLY_ASYNC *Cannot apply 'async' to this statement.*

The `async`-modifier can only be applied to **`def`**-, **`for`**- and **`with`**-statements.

CANNOT_ASSIGN_TO_CALL *You cannot assign something to a function call.*

The parser detects if the target of an assignment is a function call and reports this error. A typical example might be `foo(1) = 23`. The exact reason for this error might be due to various misconceptions. Possible reasons include an attempt to define a function or an assignment to a list with the wrong kind of brackets.

If the source is just a single name, an ASSIGNMENT_TO_RIGHT error is reported instead.

CANNOT_ASSIGN_TO_FUNCTION* *You cannot assign something to a function.*

This error is reported when a student tries to assign a value to a function from inside the functions. For instance, inside the body of the function `isPrime` we find the assignment `isPrime = False`.

Strictly speaking, overwriting even built-in functions with a value or using a local variable with the same name as the function is legal in Python. However, since it can lead to puzzling and unexpected effects for novice programmers, we report this as an error.

See also: CANNOT_REDEFINE_NAME.

CANNOT_REDEFINE_NAME* *The name 'X' is already defined.* where `X` is a name.

Some students tend to reuse names for different purposes, thereby rendering functions inaccessible. One example might be `input = input()`, where the builtin `input`-function might inadvertedly be overwritten. We found some rather subtle cases particularly in the context of parameters shadowing functions.

See also: CANNOT_ASSIGN_TO_FUNCTION.

CANNOT_TEST_TUPLE *You need to test each element of this tuple individually.*

When several variables need to be positive some students use mathematical notation and write, for instance, `if a, b > 0`. The parser will then report that Python cannot test a tuple against a number. Sometimes, however, the programmer might indeed want to compare tuples. In such instances, however, the tuple must be put into parentheses: `if (a, b) > (0, 0)`. Hence, if the parser finds such an instance, it reports the error TUPLE_NEEDS_PARENS instead.

This error is very similar to AND_CONNECTS_CMP_NOT_VARS with the only difference that the programmer used a comma instead of **`and`** to connect variables.

CANNOT_USE_KEYWORD_AS_NAME *You cannot use the keyword 'X' as a name.* where `X` is any Python keyword.

When learning to program, many students are not yet aware of all the possible keywords and might accidentally try to use a keyword as a name. The parser detects this error whenever a keyword is used as the target of an assignment or as a paramter.

CLASS_METHOD_WITHOUT_SELF *A class-method needs at least one parameter.*

Class methods in Python get a reference to the class as their first argument. A method with the `@classmethod`-decorator therefore needs to have at least one parameter.

COLON_EXPECTED *A colon ':' is required here.*

A very common, but simple error among novice Python programmers is to forget the colon in compound statements such as **`if`**, **`for`**, **`def`**, etc.

CONDITION_ALWAYS_FULFILLED* *This condition is always fulfilled.*

Similar to CONDITION_CANNOT_BE_FULFILLED the parser also detects some very simple cases where the condition is always fulfilled, e. g., `if 0 < 1`.

CONDITION_CANNOT_BE_FULFILLED* *This condition cannot be fulfilled.*

The rationale behind this error are conditions where a variable is tested to be within a given range. For instance, `if 0 < x < 3`. Either in the case where negative numbers are used or where such conditions are multiplied using copy-paste, the given range might

not contain any numbers at all. For instance, there is no number which would fulfill the condition `if 0 < x < -3`.

Even though such conditions are legal in Python and might be used consciously, they are more often source of very-hard-to-find bugs.

DECORATOR_NAME_CLASH* *A function and its decorator cannot have the same name 'X'.* where `X` is the name of the function.

When a function bears the same name as its decorator, the decorating (original) function is in fact overwritten, leading to unexpected behaviour later on and hard-to-find bugs.

For instance, TigerJython offers a decorator `@onMouseClicked` to mark a function as a callback for mouse-click events. However, some students named their own function also `onMouseClicked`, and registered several functions so as to react to mouse-click events.

DECORATOR_NEEDS_CALLABLE *Decorators are only applicable to functions and classes.*

Some students try to write code between a function and its decorator, thereby decoupling the dectorator from the function.

DEFINITION_INSIDE_LOOP* *A 'X'-definition cannot be inside a loop.* where `X` is either function or class.

Advanced Python techniques define functions with different closures inside a loop. In the context of novice programmer, however, the definition of a function or a class inside a loop is rather based on a misconception. In particular, some students put the entire program inside a loop in order to have the program being executed mutliple times in a row. Even though this might be syntactically legal, it is of little help to the novice programmer in learning to understand the concepts such as functions.

See also: IMPORT_INSIDE_LOOP.

DOUBLE_ELSE *A 'X'-structure can only have one 'else'-branch.* where `X` is **`if`**, **`while`** or any other compound keyword that has an **`else`**.

If programs grow and become larger, a programmer might accidentally define two **`else`**-branches for a condition.

See also: USE_ELIF_INSTEAD_OF_ELSE, USE_ELIF_INSTEAD_OF_ELSE_IF.

DOUBLE_EQUAL_SIGN_EXPECTED *A double equal sign '==' is required here.*

The problem where the programmer uses a single equal sign '=' for comparison is well documented and reaches far beyond Python. A typical example might be `if x = 0:`.

Standard Python usually reports that it requires a colon at the point of the single equal sign, which is almost always wrong and confusing to the novice programmer. Even though

this error might in fact be due to other omissions, a missing second equal sign is arguably the most common real cause for this error.

See also: SINGLE_EQUAL_SIGN_EXPECTED.

DOUBLE_PARAMETER_NAMES *Two parameters cannot have the same name: 'X'.* where `X` is the name of the parameter.

Corresponds to Python's message "SyntaxError: duplicate argument 'X' in function definition."

ELSE_MUST_BE_INDENTED *'X' must be indented to match the 'if'.* where `X` is either **`else`** or **`elif`**.

This is a special case of invalid indentation, where the programmer has unindented the **`else`** too much.

ELSE_WITH_COMPARISON *'else' does not have a comparison.*

Some students believe that **`else`** – like its counterpart **`if`** – requires a condition. For instance, after an `if x >= 0` they go on to write `else x < 0`.

In order to detect this error, the parser checks if the **`else`** is followed by a colon at the end of the line and a suite below. In particular, `else foo()` (without a suite) is rather missing a colon after the **`else`** and has `foo()` as the body of **`else`**.

ELSE_WITHOUT_IF *There is an 'X' without an 'if'.* where `X` is either **`else`** or **`elif`**.

Before the parser reports this error, it tries to adjust the indentation and reports either ELSE_MUST_BE_INDENTED or INDENTED_ELSE.

EMPTY_SUBSCRIPT *The subscript cannot be empty.*

In constrast to a function call without arguments, a subscript requires an index or a slice. Hence, `myList[]` is illegal.

EXTRA_INDENTATION *There is an extra indentation.*

This error indicates that a line has been indented but the preceeding statement is not a compound statement that would allow for a suite.

See also: INCONSISTENT_INDENTATION.

EXTRA_LINEBREAK *There seems to be an extra linebreak. You might want to hide it using '\'.*

In Python, a line break marks the end of a statement, even when the actual statement is intended to span several lines. Hence, when a line ends in an operator such as a plus sign and the following line is most likely an expression, the parser reports this error and joins the lines for further parsing. The error is also reported if a line is indented and begins with a dot.

EXTRA_LEFT_BRACKET *There is an extra opening bracket or parenthesis: 'X'.* where `X` is the extra bracket, e. g., `(`.

In case of the pattern ⟨*(*⟩⟨*(*⟩⟨*NAME*⟩⟨*)*⟩⟨*NEWLINE*⟩ the compiler does not try to insert the missing right parenthesis but deletes the extra left parenthesis.

See also: MISSING_RIGHT_BRACKET, EXTRA_RIGHT_BRACKET, MISSING_LEFT_BRACKET.

EXTRA_RIGHT_BRACKET *There is an extra closing bracket or parenthesis: 'X'.* where `X` is the extra bracket, i. e., `)`, `]` or `}`.

An extra right (closing) bracket could mean that somewhere there is a corresponding left bracket missing or that, indeed, there is just an extra right bracket. A common case for an extra right bracket occurs with nested bracketing as in `foo(bar(egg()))`.

See also: MISSING_RIGHT_BRACKET, MISSING_LEFT_BRACKET, EXTRA_LEFT_BRACKET.

EXTRA_SPACE *There is an extra space.*

Particularly at the beginning, novice programmers have difficulties typing program code. They first must learn that the code is case-sensitive and that spaces matter in some places, but not in others (a common question is whether there is a space between the name of a function and the following parentheses). This uncertainty leads to some novices separating the different words of a compound name such as `set pen color` instead of `setPenColor`.

When the parser detects two consecutive names, it checks whether linking these names together produces a valid name. If so, it reports that there is an extra space and removes it. If, however, both names exist at other places in the program, the parser reports an EXTRA_SPACE_OR_MISSING_COMMA error.

Finally, if the parser finds that the second name might be a field of the first one, it reports a MISSING_DOT.

See also: MISSING_SPACE.

EXTRA_SPACE_OR_MISSING_COMMA *There is an extra space or missing comma.*

This error is reported when a statement begins with two subsequent name tokens, e. g., `egg bacon()`.

See EXTRA_SPACE, MISSING_ASSIGNMENT.

EXTRA_TOKEN *There is/are extra token(s): 'X'.*

Whenever the parser fails to identify the exact cause for an error and finds that deleting one or more tokens is a viable option, it reports that there is or are extra tokens.

FOREIGN_KEYWORD *'X' is not a keyword in Python.* where `X` is a word like `to` that seems to be used as a keyword.

See FOREIGN_TOKEN.

FOREIGN_PRIVATE *The token 'X' is invalid, use underscores '_' to mark a function as 'private'.* where `X` is an invalid `private`- or `protected`-keyword.

See FOREIGN_TOKEN.

FOREIGN_STATEMENT *Python has no 'X'-statement.* where `X` is a statement-type such as `switch`.

See FOREIGN_TOKEN.

FOREIGN_SYNTAX *This X-syntax is invalid in Python.* where `X` is the name of a programming language such as `Lisp`.

If the parser detects statements such as `(let x 123)` it reports that this is invalid syntax in Python.

FOREIGN_TOKEN *The token 'X' is invalid, use 'X' instead.* where `X` are the offending token and the correct Python alternative, respectively.

Some programmers new to Python might have previous experience with other programming languages. They might then try to directly use constructs alien to Python but legal in, say, Java. Where such an intention is obvious, the parser will report an error indicating how to properly achieve the same goal in Python.

Among the different error reported are FOREIGN_TOKEN, FOREIGN_PRIVATE, FOREIGN_VAR, FOREIGN_KEYWORD, NO_END_NEEDED, FOREIGN_STATEMENT.

FOREIGN_VAR *Python does not use 'X' to define variables.* where `X` is a keyword such as `var`.

See FOREIGN_TOKEN.

FOR_TARGET_NAME_REQUIRED *The 'for'-loop requires a variable.*

Some students tried to incorporate literals directly into the `for`-loop, as in `for "e"in word`. The parser detects the use of literals in such a place and reports that a variable must be used.

FUTURE_MUST_BE_FIRST *The 'from __future__ import'-statement must be the first statement in the module.*

This error is raised if the future-import is preceded by any other statement than a docstring or another future-statement.

GENERATOR_CANNOT_RETURN_VALUE *A generator cannot use 'return' to return a value.*

As soon as a function contains a `yield`-statement, the function is turned into a generator. In such a case, however, it becomes illegal to try and return a value using `return` as all values must be returned via `yield`.

GLOBAL_MUST_BE_FIRST* *'X'-statements must be the first statements inside a function.* where `X` is `global` or `nonlocal`.

In "strict" mode the parser requires global statements to precede any other statements inside a function's body, except for docstrings, other declarations and imports. In Python, the `global`-statement can actually be used anywhere within the function, but it might lead to strange results if the variable is used before declared as global.

GLOBAL_OUTSIDE_FUNCTION *A 'X'-statement cannot occur outside a function.* where `X` is `global` or `nonlocal`.

Standard error.

IMPORT_INSIDE_LOOP* *An 'import'-statement cannot be inside a loop.*

For advanced programmers, there might be valid reasons for using an import inside a loop. In the case of novice programmers, however, this is surely sign of an error.

See also: DEFINITION_INSIDE_LOOP.

INCOMPLETE_IMPORT *This import statement is incomplete.*

An incomplete import-statement is of the form `from module import` with the name to import missing. Some novice students actually tend to omit the star at the end of such imports as they are not aware that the star bears any meaning.

INCONSISTENT_INDENTATION *The indentation is inconsistent.*

An inconsistent identation occurs when the current statement is less indented than the previous one, but not enough to signal the end of a block. Common causes are either students yet unaware that indentation is important, or a statement that would end several nested blocks with no visual connection to the proper indentation.

See also: EXTRA_INDENTATION.

INCONSISTENT_RETURNS* *This function sometimes returns a value and sometimes does not.*

Good programming practice clearly distinguishes between functions with a return value and procedures returning nothing (`None`, `Unit`, `void` or something similar). To make students aware of this distinction, the parser reports a warning if the function could be left

both by returning a value and by not returning anything (in the case of Python, the function still returns `None`). Note that for this check `return` and `return None` are distinct statements, even though their effect is the same.

More frequent cause for this error, however, is not programming style but rather a forgotten `return` in one of the branches. Think, for instance, of a function searching in a loop through a list and returning a value in case something has been found. The function should then also return something meaningful in the case the search was not successful (or raise an exception).

INDENTED_ELSE *This 'X' must not be indented.* where `X` is either `else` or `elif`.

Some students consider the `else` to be part of the `if`'s body and hence indent it accordingly. The parser detects this case and reports that the `else` should not be indented.

INFINITE_LOOP* *This is an infinite loop: it runs forever.*

Whether raising this error makes sense is certainly debatable as there are good reasons for using an infinite loop. Considering, however, that the parser was written with novice programmers in mind, an infinite loop is more often due to a forgot `break`-statement/condition.

Another cause for this error might be a student writing `while 5` in order to have Python repeat some code five times. If `repeat`-loops are enabled, the parser reports USE_REPEAT_INSTEAD_OF_WHILE in this case.

The parser does not report loops with `while True` to be infinite. In this case it assumes that the loop is supposed to run infinitely.

INITIALIZATION_INSIDE_LOOP* *You cannot have the initialization inside the loop.*

The parser checks loops for the pattern where one of the first lines assigns a value to a variable and the very same variable is updated towards the end of the loop. For instance, the first statement inside the loop might be `x = 0` and the last statement `x += 1`. Before reporting the error the parser makes sure that the variable `x` is not accessed before the initialization or after the update.

INVALID_ASSIGNMENT *You cannot assign something to 'X'.* where `X` is any expression that cannot serve as a target.

If the assignment's target expression is not a valid target expression such as a name or a subscript, the parser reports this error. Before doing so, it first checks if the source on the right-hand side is just a single variable. If so, ASSIGNMENT_TO_RIGHT is reported instead.

See also: CANNOT_ASSIGN_TO_CALL, CANNOT_ASSIGN_TO_FUNCTION, INVALID_AUGASSIGN_TARGET.

INVALID_FUNCTION_DEF *Invalid definition of a function.*

The parser tries to see, if after a `def`-keyword, a valid name and subsequently parameters are given. In case it detects a specific error such as an invalid name, it reports it and continues with parsing the function. If, however, the parser cannot identify the error and recover, it reports this error.

See also: INVALID_FUNCTION_DEF_ASSIGN

INVALID_FUNCTION_DEF_ASSIGN *Use ':' and 'return' instead of an assignment.*

This error is reported in the case of `def foo(x) = ...`.

See also: INVALID_FUNCTION_DEF.

INVALID_GENERATOR_ARG *A 'generator/comprehension' argument cannot be combined with other arguments.*

Python can use a generator/comprehension as argument to a call as, e. g., in `list(x**2 for x in range(5))`.

Such a generator argument, however, must be the only argument given and cannot be combined with other arguments.

See also: MULTIPLE_VAR_ARGS.

INVALID_INPUT_CHARACTER *This is an invalid input character: 'X'.* where `X` is an (invalid) character.

Some characters cannot be used in a Python program, in particular German umlauts and other accented characters. In such a case, the parser reports this error.

INVALID_KEY_VALUE_PAIR *This is an invalid key-value-pair.*

This error indicates that in a dictionary literal such as `{'a': 1, 'b': 2}`, either the key or the value (or both) is invalid.

INVALID_NAME *This is an invalid name: 'X'.* where `X` is the invalid name.

Particularly in the beginning, students must first learn that valid names are restricted to a subset of all characters and cannot contain, for instance, hyphens, dots or spaces. In the case of function definitions, the parser tries to read even invalid names and then reports that the name is invalid and must be corrected.

INVALID_AUGASSIGN_TARGET *This expression is not a valid target for augmented assignment.*

See INVALID_ASSIGNMENT.

INVALID_STRING_PREFIX *This is an invalid string prefix: 'X'.* where `X` is the invalid prefix.

There are only a few valid string prefixes in Python such as `r` or `b` and combinations. The actual reason for this error, however, might also be a forgotten operator.

METHOD_WITHOUT_SELF *A method requires a 'self'-parameter.*

Python requires methods to explicitely have "self" or "this" as the first parameter in methods (even though it is then not given as an argument). If a method has no parameters or the first parameter is not called "self" the parser reports this error.

Some Python libraries use "s" instead of "self" but this, again, is bad practice for novice programmers.

MISMATCHED_CLOSING_BRACKET *There are mismatched brackets or parentheses: expected 'X' but found 'X'.* where each `X` is a bracket.

Each opening/left bracket or parenthesis must be matched with a correspondings right/-closing bracket. If the two brackets do not fit together (e. g., `(]`), the parser reports this error.

In some special cases, the parser is able to determine that the two mismatched brackets are not supposed to fit together but that the error is due to malformed nesting. For instance, in the case of `([1, 2)]` the parser finds that the two closing/right brackets seems to be swapped and reports SWAPPED_TOKENS.

MISPLACED_ASSIGN *The assignment 'X' cannot occur as part of an expression.* where `X` is an assignment operator such as `=`.

Due to a misconception about assignments, some students try to write the formula $x_1 = x_0 \cdot 3 + 1$ as `(x *= 3) + 1`. In such cases, the parser will report that the assignment cannot be part of an expression.

MISSING_ASSIGNMENT *There seems to be an assignment missing.*

While parsing a statement, the parser might get stuck after having read only one token with other tokens left to be read. The usual error to report in such circumstances is the "no viable alternatives". Before reporting this error, however, the parser tries to fix the statement in a series of different attempts, given that the first token of the statement is a name.

The name beginning the statement might be a misspelled keyword (in particular **`print`**). In such a case the parser fixes the keyword and reports a MISSPELLED_KEYWORD. The name could also be merged with the following token to form a longer name. If this succeeds and the resulting name is used elsewhere in the program, the parser reports an EXTRA_SPACE.

Finally, the parser checks if the line contains an assignment operator later on. If one is found, it reports an EXTRA_SPACE_OR_MISSING_COMMA. Otherwise, if the name is a function, it reports MISSING_PARENTHESES or assumes that the assignment operator is missing.

See also: MISSING_OPERATOR_OR_COMMA.

MISSING_ASSIGNMENT_SOURCE *The assignment is missing a source expression.*

An assignment with a missing source is of the form `x =` without a value or expression on the right.

MISSING_BODY *There is a body or indentation missing.*

Compound statements such as **`if`**, **`while`** or **`def`** need a body of statements following the colon (an empty body would have to be written using **`pass`**). When the parser does not find such a body or suite, it might be because the body is indeed missing or because the body is not properly indented.

MISSING_COMMA *There is a comma missing.*

A missing comma is reported when parsing parameters or arguments.

See also: MISSING_OPERATOR_OR_COMMA.

MISSING_COMPARISON *There is a comparison missing.*

The parser will always report this error if, after an **`if`** or **`while`** there is immediately a colon following, without any test.

In "strict mode" the parser will also report this error in case a test is an expression (calculation) but has no comparison, e. g., `if x+1:` .

MISSING_DOT *There seems to be a dot missing.*

See EXTRA_SPACE.

MISSING_LEFT_BRACKET *There is a missing left bracket or parenthesis: 'X'.* where `X` is the missing bracket.

If the parser detects an extra right/closing bracket it checks if it can find a position to insert a left bracket. If not successful a EXTRA_RIGHT_BRACKET is reported. The second cause for this error is a missing left parenthesis inside a function definition.

See also: MISSING_LEFT_PARENTHESIS, EXTRA_RIGHT_BRACKET.

MISSING_LEFT_PARENTHESIS *There is a missing left parenthesis '('.*

See MISSING_LEFT_BRACKET.

MISSING_OPERATOR_OR_COMMA *There is an operator or comma missing.*

There are two situations where the parser will report this error. First, if a number is immediately followed by a name or a left parenthesis. This error is mainly due to mathematical notation creeping into programming: in math it is common to write, e. g., $2x$ instead of $2 \cdot x$, but in Python the $\star$-operator is necessary.

The second situation is inside an expression as in, e. g., `if x y == 1`. Note that the pattern `x y` as a statement is reported as a MISSING_ASSIGNMENT.

See also: MISSPELLED_NUMBER.

MISSING_PARENTHESES *There seem to be parentheses missing.*

Among the programming languages frequently used in introductory classes are not only Python and Java, but also Lisp-related languages such as Scheme or Logo. In Lisp, however, the arguments to a function are separated by space only and do not require parentheses. Hence, when the parser detects a statement starting with the name of a function and an expression following, it reports that there are parentheses missing. For instance, instead of `left 90` you should write `left(90)` in Python.

MISSING_RIGHT_BRACKET *There is a closing bracket or parenthesis missing: 'X'.* where `X` is the missing bracket.

See MISSING_LEFT_BRACKET.

MISSING_SPACE *There is a space missing.*

In cases where a keyword is concatenated with a name or number directly following the parser might detect the keyword nonetheless and report that there is a space missing. More precisely, the parser is looking for this pattern in comparisons with `in` or `is`, or for any line starting with a name that turns out to be invalid syntax. For instance, if will detect a `deffoo():`.

See also: EXTRA_SPACE, MISSPELLED_KEYWORD.

MISSING_TOKEN *Missing 'X'.* where `X` is the token missing.

This general error is reported when inserting the specific token allows the parser to continue. The parser will only report this error if no other more specific syntax error has been identified.

MISSPELLED_KEYWORD *Misspelled keyword 'X' instead of 'X'.* where `X` are the given name and the closest keyword, respectively.

If a name leads to a syntax error, the parser checks if that name could be corrected to a keyword in such a way that parsing could continue. This is mostly the case at the beginning of a line or when the grammar clearly specifies that a given keyword is expected (e. g., the `in` in `for x in list`).

In order to avoid false positives, the parser will report this error only in cases where a minor change is necessary: correcting lower-/uppercase, swapped letters, a missing or an extra letter. Some cases require the parser to take the current context into consideration: for instance, it is very difficult to distinguish between misspelled `for` and `from`.

In addition to keywords, the parser can also correct some operators. For instance, when using a binary left shift `<<` where, in fact, a comparison `<` would be expected. Again, some cases are very hard or even impossible to detect, such as, e. g., `x=-1`, which might indeed be a misspelled `x-=`.

See also MISSING_ASSIGNMENT.

MISSPELLED_NUMBER *There seems to be a typo inside your number.*

A typical example of a misspelled number is something like `123x456`.

Actually, this error might be due to more than just a typo. A novice programmer might want to use an `x` to multiply two numbers, for instance. Reporting a typo inside the number should, however, give a hint as how Python is interpreting the given tokens.

See also: MISSING_OPERATOR_OR_COMMA.

MISSPELLED_OPERATOR *Misspelled operator 'X' instead of 'X'.*

See MISSPELLED_KEYWORD.

MULTIPLE_VAR_ARGS *Only one unpacking var/keyword-argument is allowed.*

When calling a function, you can use the elements of a list or dictionary as individual arguments for the call. However, you can only use the elements of one list for this, e. g., `foo(*list1, *list2)` is invalid.

See also: INVALID_GENERATOR_ARG, MULTIPLE_VAR_PARAMS, VARARG_AFTER_KEYWORD_ARG.

MULTIPLE_VAR_PARAMS *Only one unpacking var/keyword-parameter is allowed.*

When defining a function, you can specify that all remaining positional or keyword arguments should be collected into a list or dictionary. However, there can only be one such list or dictionary.

See also: MULTIPLE_VAR_ARGS.

NAME_EXPECTED *There is a name required here.*

There are several situations that would lead to this error being reported. For instance, when in a `for`-loop the variable between `for` and `in` is missing. Of particular interest with novice programmers might be the case of parameters in a function's definition. Some students try to replace the parameters with expressions to directly assign values to them or modify the parameters. An example of this is `def foo(2*x, y)`.

NO_END_NEEDED *There is no 'end' needed or allowed in Python.*

Many programming languages use an `end` to mark the end of a block/suite. Python, however, has no marks for the end of a suite other than by indentation.

See also: FOREIGN_TOKEN, FOREIGN_SYNTAX.

NO_PARAM_DEFAULT_ALLOWED *An unpacking parameter cannot have a default value.*

Parameters in a function's definition can have default values assigned to them. However, the parameters for collecting all remaining positional and keyword arguments, respectively, cannot have such default values.

NO_VIABLE_ALTERNATIVE *There is no viable alternative at 'X'.* where `X` is the token where parsing is not possible anymore.

This is one of the most generic and least frequent error messages reported. The parser reports no viable alternative only if it was unable to identify any other syntax error, but cannot continue with parsing at the present position. As a fallback message, the situation where it is actually reported depends on all the other syntax errors and might vary as the parser is further developed.

NUMBER_NOT_SUBSCRIPTABLE *A number cannot have a subscript.*

The cause for this error message is unlikely to be an attempt to use a subscript on a number. Rather, there might be a missing operator or comma. It could also be the cause of foreign syntax, as square brackets are used in "Logo" in situations like `repeat 4 [fd 100 lt 90]`. The parser reports this error, nonetheless, as it gives the student a direct feedback about how the given tokens are interpreted in Python.

See also: FOREIGN_SYNTAX.

PARAM_AFTER_KEYWORD_PARAM *The unpacking keyword-parameter must come last.*

Python allows a special parameter to collect all remaining keyword arguments. This parameter `**kwargs`, however, must come last of all parameters.

PARAMS_REQUIRED *Parameter(s) required but 'X' found.* where `X` is a token, mostly a colon.

When a function does not have any parameters, both the definition and the call still require empty parentheses. In case of a function definition without parentheses, the parser reports this error.

POS_ARG_AFTER_KEYWORD *Positional arguments cannot follow keyword arguments.*

When calling a function, all positional arguments must come before any keyword argument.

See also: POS_PARAM_AFTER_KEYWORD.

POS_PARAM_AFTER_KEYWORD *Parameters without defaults cannot follow parameters with default value or unpacking parameters.*

When definition a function, mandatory parameters (those without default value) must precede other parameters such as those with default value or varargs.

See also: POS_ARG_AFTER_KEYWORD.

PRINT_DEST_EXPECTED *'>>' must be followed by a valid output destination.*

In Python 2 you can specify the destination of a `print`-statement via the `>>`-operator. When this operator is present, it requires a valid destination (while the `print`-statement itself does not need any arguments).

PRINT_IS_STATEMENT *In Python 2.x 'print' is a statement and cannot be called with keyword arguments.*

The is the counterpart to PRINT_NEEDS_PARENTHESES. It is reported when print is treated as a function in Python 2.

See also: PRINT_NEEDS_PARENTHESES.

PRINT_NEEDS_PARENTHESES *In Python 3.x 'print' is a function and requires parentheses.*

One of the most obvious differences between Python 2 and 3 is that `print` has become a function instead of a statement, hence requiring parentheses. Since there are many tutorials for both versions of Python, the parser provides this hint.

PYTHON_2_FEATURE_NOT_AVAILABLE *This feature from Python 2.x is not available.*

While Python 2 allows parameters to be tuples of names, this feature has been removed in Python 3. Accordingly, `def foo(a, (b, c))` is legal in Python 2, but not in Python 3.

PYTHON_3_FEATURE_NOT_AVAILABLE *This feature from Python 3.x is not available.*

Python 3 has extended the syntax for unpacking using star-notation. The parser recognizes such syntax and reports that this is not available in Python 2.

RETURN_OUTSIDE_FUNCTION *A 'return'-statement cannot occur outside a function.*

Standard error.

See also: USE_BREAK_INSTEAD_OF_RETURN.

SINGLE_EQUAL_SIGN_EXPECTED *Use a single equal sign '=' for assignment.*

Just as some students use a single equal sign for comparisons, some then wrongly use the double equal sign for assignment. Hence, the parser reports this error upon detecting the pattern where a statement is just a comparison for equality with a name on the left hand side.

SEE ALSO: DOUBLE_EQUAL_SIGN_EXPECTED

SUPERFLUOUS_COMPARISON *The comparison to 'X' is superfluous here.* where `X` is either `True` or `False`.

Some students write test always with a comparison to `True` and `False`, such as `if x > 0 == True`. The parser detects these tests and reports that these comparisons are not necessary or that the student should use **`not`**.

See also: USE_NOT_INSTEAD_OF_FALSE.

SWAPPED_TOKENS *These tokens seem to be swapped: 'X' and 'X'.* where `X` denote two tokens, mostly two brackets.

The parser tries to make sure that brackets are correct at an early stage in the parsing process. If possible, missing or extra brackets are inserted or removed, respectively. But the parser also detects patterns such as `([...)]` with two of the brackets swapped. In such cases, the parser reports the swapped brackets instead of inserting/deleting.

TOKEN_REQUIRED *'X' required but 'X' found.* where `X` are two tokens.

This is message is reported when a specific token is required but something else has been found.

TUPLE_NEEDS_PARENS *This tuple needs to be enclosed in parentheses.*

See: CANNOT_TEST_TUPLE.

UNEXPECTED_END_OF_INPUT *Unexpected end of line or input.*

This error is reported whenever the parser requires further input to complete parsing the current statement. An instance of such a case is `x = 1 +` with a missing second/right operand.

UNEXPECTED_KEYWORD *The keyword 'X' cannot occur at this point.* where `X` is a keyword.

Similar to ELSE_WITHOUT_IF this error is reported when a keyword such as **`finally`** occurs outside of a valid context, i. e., without the preceding **`try`**.

UNREACHABLE_CODE *This code is unreachable and will never be executed.*

Keywords such as **`return`**, **`break`** or **`continue`** cause the current block of code to be left immediately. Students, however, often assume that, before actually exiting a loop, say, Python will still execute the statements following the **`break`**-statement. Hence, the parser reports code after an exiting statement as unreachable.

UNTERMINATED_STRING *This string is unterminated.*

If a string literal contains line breaks, the parser reports that the string has not been terminated. This does not apply, of course, for multi-line strings starting with triple quotes.

USE_AND_NOT_COMMA *Multiple comparisons are combined by 'and' or 'or' instead of a comma.*

The parser tries to carefully examine tests in comparisons. If it finds that a comparison is followed by a comma as in, e. g., `if a > 0, b > 0` it will report this error.

See also: USE_COMMA_NOT_AND.

USE_BREAK_INSTEAD_OF_RETURN *Use 'break' instead of 'return' to exit a loop.*

Students sometimes confuse **`break`** and **`return`** and use one for the other. In the case where a **`break`**-statement is used outside a loop but inside a function, the parser reports a USE_RETURN_INSTEAD_OF_BREAK. Likewise in the case of a **`return`**-statement inside a loop but not inside a function, the parser reports this error.

See also: USE_RETURN_INSTEAD_OF_BREAK, RETURN_OUTSIDE_FUNCTION, BREAK_OUTSIDE_LOOP.

USE_COMMA_NOT_AND *Multiple values are separated by comma instead of 'and'.*

It has long been reported that the english words used for keywords sometimes suggest a wrong meaning to novice programmers. One such instance is where a novice programmer writes `return a and b` in order to return both values a and b. Unless the parser finds these values to be Booleans, it will report that the programmer should rather use a comma instead of **`and`** in this case.

See also: USE_AND_NOT_COMMA.

USE_ELIF_INSTEAD_OF_ELSE *Use 'elif' instead of 'else'.*

The parser will report when an **`if`**-statement has more than one **`else`**-branch. However, if the first **`else`**-branch also has a comparison following it, the parser assumes that the **`else`** should indeed be an **`elif`** instead.

See also: USE_ELIF_INSTEAD_OF_ELSE_IF, DOUBLE_ELSE, ELSE_WITH_COMPARISON.

USE_ELIF_INSTEAD_OF_ELSE_IF *Use 'elif' instead of 'else if'.*

Programming languages other than Python often use **`else if`** instead of a dedicated **`elif`**. To help programmers coming from such languages, the parser reports that an **`else if`** must be written as **`elif`**.

See also: USE_ELIF_INSTEAD_OF_ELSE, DOUBLE_ELSE.

USE_EQ_INSTEAD_OF_NEQ* *Use '== X' instead of '!= X'.* where `X` is `True` and `False`, respectively.

As students tend to use convoluted constructs, the parser tries to point some of these out and help the student in finding a simpler alternative.

USE_MOD_NOT_DIV *Use '%' instead of '/' to check for divisibility.*

Modulo/remainder operation seems to be an inherently difficult topic for students. They frequently end up writing `if x / 2 == 0` instead of `if x % 2 == 0`. Dividing a variable and then checking if the result is zero or non-zero does not make sense as one could test the variable x directly. The parser therefore warns that the programmer probably intended to use the remainder operator instead of a division.

USE_NOT_INSTEAD_OF_FALSE* *Use 'not' rather than a comparison to 'X'.* where `X` is either `True` or `False`.

See: SUPERFLUOUS_COMPARISON.

USE_REPEAT_INSTEAD_OF_WHILE* *Use 'repeat' instead of 'while'.*

We have extended Python by the keyword `repeat` as a simple looping structure for repeating the code a given number of times. One could then write `repeat 3: print "."` to have three dots printed. As some students have already heard that a loop is done using `while`, they then go on to write something like `while (3)`. Hence, the parser checks for `while`-tests consisting of only a single number and then reports that the programmer should rather use `repeat`.

USE_RETURN_INSTEAD_OF_BREAK *Use 'return' instead of 'break' to exit a function.*

See USE_BREAK_INSTEAD_OF_RETURN.

USELESS_COMPUTATION* *The result of this expression is never used.*

While a useless statement has no side effects at all, a useless computation might actually include function calls. However, the overall statement is an expression whose result is not used, e. g., `2 * foo()`.

See also: USELESS_STATEMENT.

USELESS_STATEMENT* *This statement is useless: it has no effect.*

Python features an interactive console which allows to quickly evaluate expressions such as `x * 2`. Such expressions, however, do not make sense in a script/program. The parser therefore points out that expressions without side effects are useless, even though they are actually valid in Python. A special case are identifiers, which might be intended to call a function.

In fact, the intent behind `x * 2` might be to double the value of x. Therefore, if the parser detects a useless statement beginning with a name and a subsequent operation, it proposes to turn the operation into an augmented assignment. In this case that would be `x *= 2`.

See also: CALL_NEEDS_PARENTHESES, USELESS_COMPUTATION.

USELESS_STMT_USE_AUG_ASSIGN* *This statement is useless. Did you mean 'X='?*

See USELESS_STATEMENT.

VARARG_AFTER_KEYWORD_ARG *The unpacking var-argument must precede the unpacking keyword-argument.*

Python allows for lists and dictionaries to be unpacked so that their elements can act as individual arguments to a function. For instance, you might write `foo(*mylist, **mydict)`. As with positional and keyword-arguments, the list unpacking must precede the dictionary unpacking in such a case.

VARARG_NOT_ALLOWED *Unpacking var/keyword-arguments are not allowed at this point.*

This error is only reported in Python 3 when trying to use a star inside the bases of a class.

WRONG_BRACKET *Wrong parenthesis or bracket: 'X' is required instead of 'X'.* where `X` are two kinds of brackets.

If the programmer clearly uses the wrong brackets as in, e. g., `def foo[x]`, the parser will report this error.

WRONG_TOKEN *Wrong symbol 'X' instead of 'X'.* where `X` are the wrong and the correct token.

This errors is reported when, e. g., a comma is required, but a dot found, and replacing the tokens fixes the problem. In contrast to TOKEN_REQUIRED, the correct token is not required but a viable option at this point.

YIELD_OUTSIDE_FUNCTION *A 'yield'-expression cannot occur outside a function.*

Standard error.

Python's Official Grammar

For the reader's convenience we reprint Python's official grammar as can be found in [48], chapter 9 "Full Grammar specification". We have omitted some annotations that are irrelevant for this thesis.

```
# Grammar for Python

# [...]

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() and input() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')'
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
fpdef: NAME | '(' fplist ')'
fplist: fpdef (',' fpdef)* [',']

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal assignments, additional restrictions enforced by the interpreter
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
```

```
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['," test ['," test]]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' ('.'* dotted_name | '.'+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
exec_stmt: 'exec' expr ['in' test [',' test]]
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt
             | funcdef | classdef | decorated
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test [('as' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

# Backward compatibility cruft to support:
# [ x for x in lambda: True, lambda: False if x() ]
# even while also allowing:
# lambda x: 5 if x else 2
# (But not a mix of the two)
testlist_safe: old_test [(',' old_test)+ [',']]
old_test: or_test | old_lambdef
old_lambdef: 'lambda' [varargslist] ':' old_test

test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
```

```
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
listmaker: test ( list_for | (',' test)* [','] )
testlist_comp: test ( comp_for | (',' test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
                  (test (comp_for | (',' test)* [','])) )

classdef: 'class' NAME ['(' [testlist] ')'] ':' suite

arglist: (argument ',')* (argument [','
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
argument: test [comp_for] | test '=' test

list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]

comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]

testlist1: test (',' test)*

# not used in grammar, but may appear in "node < sß spassed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [testlist]
```

Bibliography

[1] A. Altadmri and N. C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, 2015. ACM.

[2] H. Amann and J. Escher. *Analysis I*. Birkhäuser, 1998.

[3] J. Arnold, T. Kohn, and A. Plüss. Programming concepts in Python with TigerJython. `http://www.tigerjython.ch/engl/index.php`, 2016. Accessed 2016-11-27.

[4] P. Bayman and R. E. Mayer. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Commun. ACM*, 26(9):677–679, Sept. 1983.

[5] B. A. Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 126–131, New York, NY, USA, 2016. ACM.

[6] H.-J. Böckenhauer and J. Hromkovič. *Formale Sprachen*. Springer Vieweg, 2013.

[7] E. Börger and R. Stärk. *Abstract State Machines*. Springer, 2003.

[8] N. C. Brown and A. Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[9] P. Byckling and J. Sajaniemi. Roles of variables and programming skills improvement. *SIGCSE Bull.*, 38(1):413–417, Mar. 2006.

[10] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, Mar. 1964.

[11] P. Denny, A. Luxton-Reilly, and D. Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pages 273–278, New York, NY, USA, 2014. ACM.

[12] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. ACM.

[13] B. Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2:57–73, 1986.

[14] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box. *Int. J. Hum.-Comput. Stud.*, 51(2):265–277, Aug. 1999.

[15] L. Grandell, M. Peltomäki, R.-J. Back, and T. Salakoski. Why complicate things?: Introducing programming in high school using Python. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 71–80, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[16] P. Gross and K. Powers. Evaluating assessments of novice programming environments. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 99–110, New York, NY, USA, 2005. ACM.

[17] D. Grune and C. J. H. Jacobs. *Parsing Techniques. A practical guide*. Springer, 2008.

[18] P. J. Guo. Python is now the most popular introductory teaching language at top U.S. universities. `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u fulltext`. Accessed: 2016-10-21.

[19] P. J. Guo. Online python tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[20] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, July 2000.

[21] L. J. Halbeisen. *Cimbinatorial Set Theory*. Springer, 2012.

[22] A. Heck. Variables in computer algebra, mathematics, and science. In *International Journal of Computers in Mathematics Education, 8 No*, pages 195–221, 2001.

[23] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[24] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156, Jan. 2003.

[25] J. Hromkovič. *Einführung in die Programmierung mit LOGO*. Vieweg+Teubner, 2010.

[26] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Proceedings Frontiers in Education 35th Annual Conference*, pages T4C–T4C, Oct 2005.

[27] K. Jensen and N. Wirth. *PASCAL User Manual and Report*. Springer-Verlag New York, Inc., New York, NY, USA, 1974.

[28] The Jython project. `http://www.jython.org/`, 2016. Accessed: 2016-11-19.

[29] T. Kohn. TigerJython. `http://jython.tobiaskohn.ch/`, 2016. Accessed 2016-11-27.

[30] M. Kölling. The greenfoot programming environment. *Trans. Comput. Educ.*, 10(4):14:1–14:21, Nov. 2010.

[31] M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):182–196, December 2003.

[32] M. Kuittinen and J. Sajaniemi. Teaching roles of variables in elementary programming courses. *SIGCSE Bull.*, 36(3):57–61, June 2004.

[33] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. *ITiCSE '05 Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 14–18, 2005.

[34] R. Lister. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 9–18, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

[35] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150, June 2004.

[36] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, 39(1):499–503, Mar. 2007.

[37] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1):57–80, 2011.

[38] B. Manaris and A. R. Brown. *Making Music with Computers: Creative Programming in Python*. CRC Press, 2014.

[39] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM.

[40] L. McIver and D. Conway. Seven deadly sins of introductory programming language design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, SEEP '96, pages 309–, Washington, DC, USA, 1996. IEEE Computer Society.

[41] I. Milne and G. Rowe. Difficulties in learning and teaching programming – views of students and tutors. *Education and Information Technologies*, 7(1):55–66, Mar. 2002.

[42] D. Muller, J. Bewes, M. Sharma, and P. Reimann. Saying the wrong thing: improving learning with multimedia by including misconceptions. *Journal of Computer Assisted Learning*, 24(2):144–155, 2008.

[43] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 168–172, New York, NY, USA, 2008. ACM.

[44] U. Nikula, J. Sajaniemi, M. Tedre, and S. Wray. Python and roles of variables in introductory programming: Experiences from three educational institutions. *JITE*, 6:199–214, 2007.

[45] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223, Dec. 2007.

[46] A. Plüss. aplu.ch. `http://www.aplu.ch/home/apluhomex.jsp`, 2016. Accessed 2016-11-19.

[47] R. T. Putnam, D. Sleeman, J. A. Baxter, and L. K. Kuspa. A summary of misconceptions of high school Basic programmers. *Journal of Educational Computing Research*, 2(4):459–472, 1986.

[48] Python Software Foundation. The Python Language Reference. `https://docs.python.org/2/reference/index.html`, 2016. Accessed: 2016-09-29.

[49] P. Reutemann. Advanced data mining with Weka. `http://www.cs.waikato.ac.nz/ml/weka/mooc/advanceddataminingwithweka/slides/Class5-AdvancedDataMiningWithWeka-2016.pdf`. Accessed: 2016-11-27.

[50] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: a review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[51] G. V. Rossum and F. L. J. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

[52] R. Samurçay. The concept of variable in programming—its meaning and use in problem-solving. *Educational Studies in Mathematics*, 16(2):143–161, 1985.

[53] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[54] E. Soloway and J. C. Spohrer. *Studying the Novice Programmer*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1988.

[55] W. A. Stein. Sage mathematics software. `http://www.sagemath.org/`, 2016. Accessed: 2016-11-19.

[56] W. Toomey. Quantifying the incidence of novice programmer's errors. `http://minnie.tuhs.org/Programs/BlueJErrors/arjen_draft.pdf`, 2011. Accessed: 2016-09-29.

[57] Wikipedia. Brainfuck. `https://en.wikipedia.org/wiki/Brainfuck`, 2016. Accessed: 2016-11-19.

[58] N. Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.

[59] N. Wirth. *Systematisches Programmieren: Eine Einführung*. Leitfäden der angewandten Mathematik und Mechanik - Teubner Studienbücher. Vieweg+Teubner Verlag, 1993.