A System for Classifying and Clarifying Python Syntax Errors for Educational Purposes

by

Anne K Kelley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by . . . .
Adam Hartz
Lecturer
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chairman, Department Committee on Graduate Theses

A System for Classifying and Clarifying Python Syntax Errors for Educational Purposes

by

Anne K Kelley

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Many students learning Python frequently encounter syntax errors but have difficulty understanding the error messages and correcting their code. In this thesis, we designed, implemented, and performed preliminary testing of a system for classification of syntax errors commonly made by beginning coders. Errors are classified by constructing a partial syntax tree and analyzing the node containing the error with respect to its surrounding nodes. The system aims to use the classified errors to provide more precise and instructive error messages to aid students in the debugging process.

Thesis Supervisor: Adam Hartz
Title: Lecturer

Acknowledgments

I would like to thank Mr. Hartz for his guidance, endless ideas, and enthusiasm through every stage of the project. His willingness to talk through ideas, make suggestions, and patience had a huge impact on the success of the project.

Additionally, I'd like to thank Samantha Briasco-Stewart, another MEng student, who developed for her a thesis a user interface that pairs with the classification process described in this thesis to build a more complete tool.

Finally, I'd like to thank Rob Miller for his assistance in the approval process for human subject experimentation and my sister, Elizabeth, for her help with editing and formatting.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Many novice programmers struggle to debug their own code because the error messages in most coding environments are geared towards expert users and demand familiarity with technical jargon and programming principles. As a result, many find the debugging process to be opaque and frustrating. This is especially problematic because novices tend to frequently encounter error messages; in a study done by Denny et al. nearly half of code submissions by students in the top quartile of a beginning Java class contained syntax errors and therefore could not even compile [9]. Furthermore, about two thirds of these top students had at least four sequential syntactical errors in their submissions, suggesting that typical error messages were not enough to help them debug their code [9].

Ideally, students would see an error message and then be able to think critically about their code, add print statements as needed, and arrive at correct code quickly and without assistance. One of the challenges of enhancing error messages is constructing messages that help build debugging skill instead of simply help fix the problem at hand. This is important, because, as seems intuitive, students with better debugging skill also tend to have superior programming performance to those who lack skill in debugging [6].

To address this situation, we aimed to develop a tool to provide students with clearer and easier to understand error messages. In an analysis of error messages from two coding platforms, one in Python and the other in Java, Pritchard gathered evidence suggesting that a small number of unique errors dominate the bulk of errors that students encounter [1]. Both platforms analyzed in his study were large, publicly-available coding platforms targeted at beginners that allowed participants to run any code of their choice [1]. This suggests that the results are likely to closely resemble the error distribution of our target population of novice coders, so we should be able to construct a limited set of error messages covering the bulk of errors that students encounter.

1.1.1 Python and the IDLE Interface

While novices encounter error messages in essentially every language, we chose to focus on Python because its use as a tool to teach fundamentals of computer science is ubiquitous. It is commonly one of the first, if not the first, programming languages that students encounter since over two thirds of the top 39 undergraduate computer science programs use Python in their first course [4]. In particular, at MIT the introductory computer science classes such as 6.01 and 6.145 are taught in Python. Often office hours are full of students who are learning to code for the first time asking for help with finding and correcting errors in their Python code.

Furthermore, many introductory courses utilize IDLE, the default open-source IDE packaged with Python releases. As a result, Python and IDLE were natural choices to focus on developing a tool in order to assist students.

1.1.2 Syntax Errors

One of the primary design decisions in creating a tool to help students interpret error messages is which errors to address.

Some early tools, such as Expresso, designed at Bryn Mawr in 2003 to assist with Java programming, relied on faculty and TA input to determine which common student errors would be helpful to address [5]. However, in similar work later done at West Point, the list generated by faculty ended up not accurately capturing the difficulties that students were having [2]. As a result, it is clear that it is necessary to rely on quantitative measurements of the difficulties students have instead of qualitative observations from staff.

To this end, we analyzed a data set of coding samples from beginners at MIT and also explored a similar data set available from past research to determine the prevalence of different types of errors.

MIT Dataset

In order to determine the pattern of errors made by MIT students in introductory programming classes, we analyzed a data set of submissions to the online tutor for the January 2017 version of 6.145, a month-long introductory class, and Spring and Fall 2016 versions of 6.01, a semester-long introductory class. Over these three classes, we counted the total number of submissions that contained each type of error.

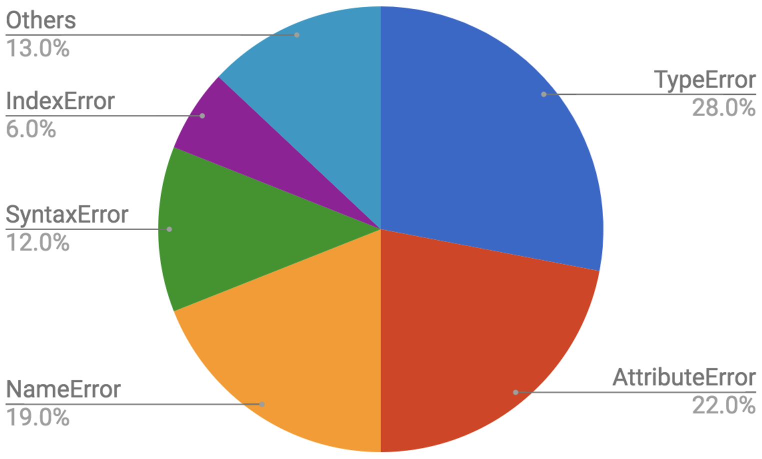


Figure 1-1: Total Distribution of MIT Error Messages

Table 1.1: MIT Python Error Message Distribution by Class

	6.145 IAP 2017	6.01 Spring 2016	6.01 Fall 2016	Overall
TypeError	32%	27%	29%	28%
AttributeError	22%	23%	19%	22%
NameError	16%	19%	21%	19%
SyntaxError	8%	14%	11%	12%
IndexError	8%	4%	7%	6%

The MIT data set confirms that novice programmers often encounter errors. Of the submissions, 66% in 6.145 resulted in an error while 44% in each of the 6.01 classes resulted in an error. The top five types of errors in the entire data set is displayed above in figure 1-1, and a detailed breakdown between classes is in table 1-1.

Note that the distribution of errors is consistent across the three classes. It is important to keep in mind, though, that students in both 6.145 and 6.01 are encouraged to run and debug their code on their own machines before submitting it to the online tutor. Consequently, the distribution of errors in the analysis above is likely skewed compared to the distribution of errors that the students are actually experiencing as they work on their code.

Interestingly, assignment type can influence the distribution of errors even within a single introductory course. We broke down the Spring 2016 6.01 class into three different types of assignments:

1. Design Labs - 3 hour long team projects
2. Software Labs - 1.5 hour long individual projects
3. Exercises - Week long homework assignments that students complete on their own time

The distributions within these three types of assignments are displayed in figures 1-2, 1-3, and 1-4.

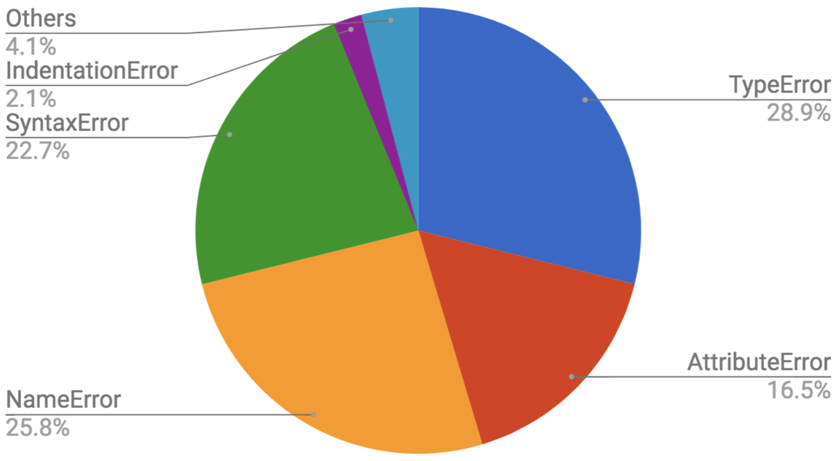


Figure 1-2: Error Distribution for 6.01 Design Labs

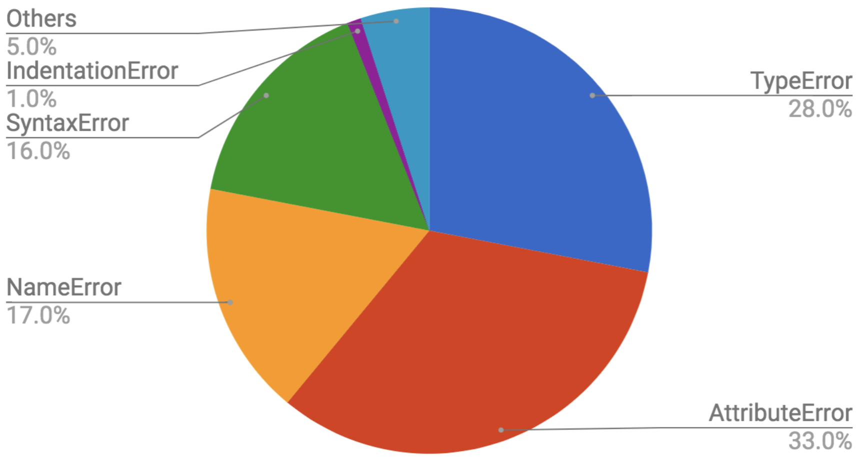


Figure 1-3: Error Distribution for 6.01 Software Labs

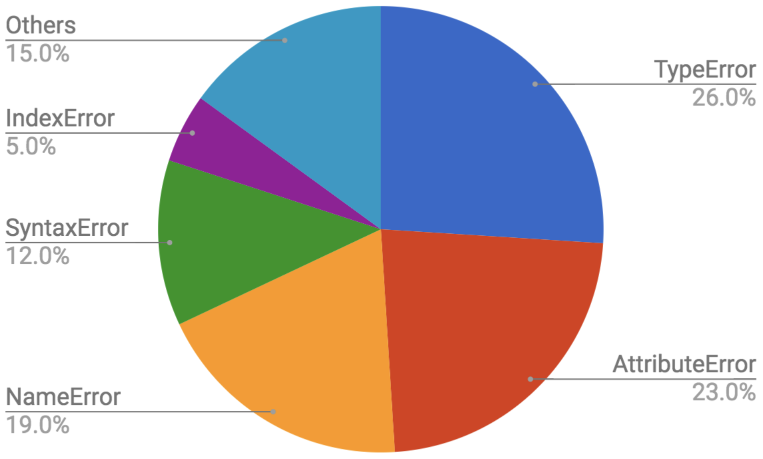


Figure 1-4: Error Distribution for 6.01 Exercises

Students make significantly more syntax errors on design labs than they do on individual exercises. It is possible that this is a result of students experiencing time pressure being more likely to submit their code directly to the online tutor, instead of running and debugging it locally. If this is the cause of the discrepancy between assignments, the distribution of errors encountered during design labs might more closely resemble the distribution of total errors students are actually making. This would suggest that students are encountering more syntax errors than observed through submissions to the online tutor.

Another limitation of the way this analysis was performed was that it counted the number of submissions that had each error and did not correct for the fact that a few students might be responsible for the majority of errors and consequently skew the average. To correct for this, we recorded the number of syntax errors encountered in all attempts for a particular problem by a single student in the 6.01 data set. We then calculated statistics based off of these unique pairs of student and assignment.

Of unique student and assignment pairs, 18.6% had at least one submission that contained a syntax error. On average, each student and assignment pair encountered around 0.39 syntax errors and the largest number of syntax errors attributed to any one student on a single assignment was 29. This analysis both confirms the suspicion that some students repeatedly submit submissions with syntax errors for the same assignment, but further implies that many students are frequently encountering syntax errors. The full distribution of errors is displayed in figure 1-5.

Altogether, this suggests that syntax errors are both prevalent among the beginning coders at MIT and that in many cases students have a difficult time debugging their code on the basis of the current error messages, leading to repeated syntactically invalid submissions.

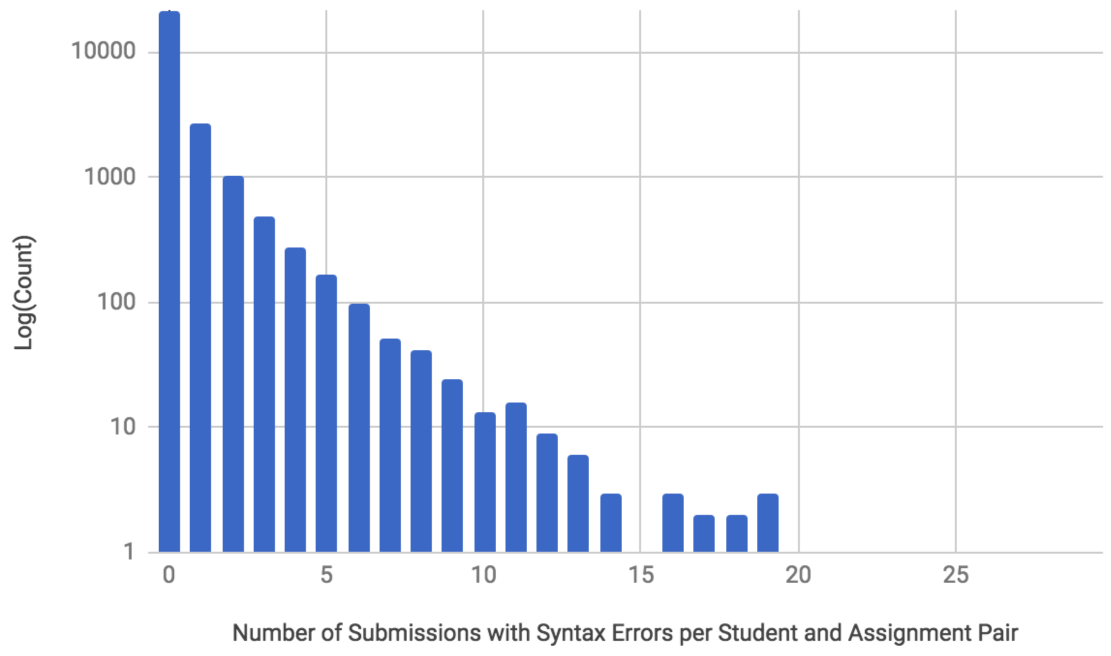


Figure 1-5: Histogram of Syntax Errors Encountered by Student and Assignment Pairs

Data Sets from Literature

A similar study was performed by Pritchard, who analyzed errors produced in CS Circles, the online Python learning environment previously mentioned. He determined that a small set of errors were by far the most common. In particular, he found that errors were dominated by the following [1]:

1. 28% - Syntax Error: Invalid Syntax
2. 15% - NameError: name is not defined
3. 12% - EOFError: EOF when reading a line
4. 4% - SyntaxError: unexpected EOF while parsing
5. 3% - IndentationError: unindent does not match any outer indentation level

Notably, the MIT distribution of errors differs significantly from what Pritchard found in his analysis of CS Circles which is plotted in figure 1-6 [1]. In particular, Pritchard's data set suggests that syntax errors are even more prevalent than was

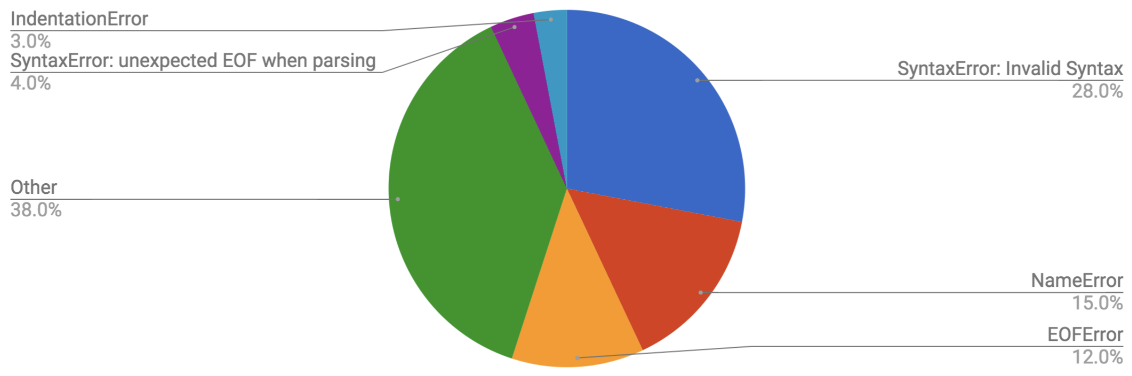


Figure 1-6: Distribution of CS Circles Python Error Messages [1]

evident from the MIT data set. However, it is possible that the difference between the MIT data and the literature's data stems from a difference in population, or a difference in the types of assignments the students are asked to complete. One principle difference is that unlike the MIT courses, students are not encouraged to run and debug their code locally before submitting to CS Circles. Consequently, the data from CS Circles might more accurately be capturing the errors that students are encountering. This would suggest that syntax errors are more prevalent than was measured in the MIT data set.

Together, the literature and the MIT data set suggest good candidates for errors that our tool should address. In particular, we targeted syntax errors since the literature shows that in some cases they are the most common error and the MIT data set confirms that they occur frequently within our target population.

1.1.3 Specificity of Error Messages

One factor that influences how useful an error message is to a novice is how much information it provides. There are a wide range of error messages built into standard Python and some provide much more feedback about the mistake than others.

Consider, for example, "NameError: name `'x'` is not defined". In this case, the error message informs the user that the variable `'x'` has not been defined in their code. This not only avoids the use of specific jargon, but it provides a path forward since it lets the student know that to correct the error they need to define `'x'`.

Another fairly specific Python error is "IndexError: string index out of range". While this message does utilize some programming specific jargon, it also informs the user of exactly what problem occurred. With this information a student could potentially add a print statement to see what index they are attempting to use that is out of range.

On the other hand, in some circumstances Python produces error messages which provide must less specific information. Consider, for example, an error message of "SyntaxError: Invalid Syntax". In contrast to the previous messages, this message does not provide any specific information on what went wrong or any potential next steps for debugging. The particularly vague nature of this error can make it difficult for novices to address.

1.2 Past Work

Several tools already exist that aim to provide more instructive error messages to students. While many of these past tools center around C++ and Java, which previously were typical introductory programming languages, they still provide valuable insight for the development of a tool in Python. In particular, they suggest which types of errors are the most common, how to construct the infrastructure to analyze errors, and what is still unknown about the most beneficial way of augmenting error messages.

One such tool to augment error messages was built by Ahmadzadeh and collaborators. In their study, a Java compiler was augmented to collect error messages, time stamps, and source code in order to determine what errors novices were encountering. Importantly, this work confirmed that that the error messages staff thought students were encountering most often did not follow the actual pattern of compiler errors the students were making [6]. Further tools have since been developed using similar paradigms of analyzing actual errors in student code, such as Gauntlet [8] and Arjen [7].

Another key design decision is how a tool to address these errors should be designed and when the assessment of the code should be performed. There are a variety of possibilities, including pre-processing before compilation, post-processing to translate error messages, or dynamic interpretation that provides real-time feedback. Among the tools previously discussed, a variety of these options have been utilized: Expresso was implemented as a pre-processor [5], Gauntlet was integrated into an existing IDE [8], and Arjen sent source code to a server where analysis was run, although its developers noted that it could have been integrated directly into an IDE [7]. The trade-offs here are numerous. Designing a system such as Arjen with a server-side component makes it easier to aggregate data from users, although it has the disadvantage of requiring an internet connection. From previous work it is not clear which setup is the ideal choice, but evidence suggests that all are feasible.

Another relevant question that has been studied in the past is the best way to enhance error messages to help beginner programmers to learn. Some studies suggest that simply providing more detail does not help students [3]. Arjen provided an example of code that could have caused the error as well as how to correct the error that it represented [7]. However, work by Denny et al. showed no effect of enhanced error messages containing line numbers, details, and example code fragments with proposed fixes [10]. They hypothesized that this might be a result of students not reading their new longer messages. Therefore what sorts of information a tool should provide in order to best assist students remains an open, and important, question.

1.3 Outline

This paper discusses a project to enhance Python's IDLE interface so that it provides more useful and informative error messages alongside standard Python output to help beginners understand their errors, learn to interpret the built-in messages, and develop their coding abilities. This paper describes the algorithms used to parse students' code in the presence of potential syntax errors, traverse the partial syntax tree and determine the likely cause of the students' errors, and report this information back to the student. It also details extensions that attempt to automatically correct the students' errors and to correct multiple errors at the same time.

Chapter two describes the desired specifications of a debugging tool, the motivations for the design decisions made, and the implementation details.

Chapter three describes extensions to the base system to allow multiple errors to be corrected at the same time and an autocorrect feature.

Chapter four contains results from preliminary testing of the coverage and accuracy of the tool on student code samples.

Chapter five is a discussion of the results and directions for future work.

Chapter 2

System Design

This chapter of the paper introduces the overall objectives and design goals of the tool. Then, it overviews the system design before diving into the implementation details necessary in order to build the system.

2.1 Introduction

2.1.1 Objectives

The end-goal of the project was to develop an add-on module to the existing IDLE interface that provides enhanced error messages. This goal can be divided into three subgoals: increasing granularity, improving error locality, and constructing instructive error messages.

Increasing Granularity

Currently there are many situations in which Python provides extremely vague error messages in the presence of syntax errors. For example, consider the following code snippet:

```
x = 4
if x == 4
    print('cat')
```

When run through standard Python, this produces an error message which reads "SyntaxError: Invalid Syntax". Ideally, we would like to provide more specific information about the cause of the syntax error. In this particular example, it would be useful to point out that every conditional statement must end in a colon, which is missing in the above code.

Improving Error Locality

The current implementation of Python attempts to pinpoint where the error occurred, but sometimes further confuses students by instead highlighting where it realized there was an error which can be several lines after the source of the error. Consider, for example, the following code snippet:

```
x = 2
x1 = ((x-3)/4)
x2 = ((x+4)/5) )
```

Here, the error is a missing closing parenthesis on the line defining `x1`, but the highlighted character is instead the closing parenthesis on the line defining `x2`. Ideally we would instead pinpoint the exact part of a line that needs to be changed.

Constructing Instructive Error Messages

Ultimately, the goal of the new error messages is to help the student to narrow down where their error occurred and think about what might have gone wrong instead of telling them explicitly what they need to change. One of the challenges of creating more informative error messages is balancing the desire to be helpful to the student with the need to not just tell the student what to do. Much like when helping a student in person, the idea is that the student will be better prepared to debug in the future if they have a greater understanding of what went wrong rather than simply being told what to change. Ideally the embellished error messages will be carefully worded to help students learn how to debug.

2.1.2 Design Goals

The process of building a system to meet the above objectives was guided by the following design goals:

- Correctness - It is very important that the more detailed explanations are never wrong, since incorrect error messages might make students less likely to discover their mistake than they would have been otherwise. To this end, we would rather provide no elaboration over Python's built-in error message than provide inappropriate or incomprehensible elaboration.

- Modularity - Since the tool is performing checks for a large number of potential errors we wanted to construct it in such a way that each check exists as its own module. This is important because it allows us to tweak our checks for individual errors without having to worry about unintended consequences. Another advantage of building a modular tool is that it could be embedded in other IDEs beyond IDLE in the future.

Common design goals which we are not concerned with include performance and scalability with input size. These are not very important because the tool is aimed at beginners working with small code bases. As a result, optimizing the run time is not essential because in most cases we don't expect algorithmic bottlenecks. In particular, our tool works by first attempting to parse code in the standard way, and then re-parsing and analyzing the result if a syntax error is encountered .

2.2 System Design Overview

The core of the project is composed of three layers. The base layer of the system is standard IDLE using the traditional Python parser. Above this is a control layer which manages the assignment of an enhanced error message. The top layer is the module layer which is comprised of an ordered sequence of modules which each test for a particular syntax error. These three layers, the interfaces between them, and their connection with Parso, which creates the partial abstract syntax tree, are depicted in figure 2-1.

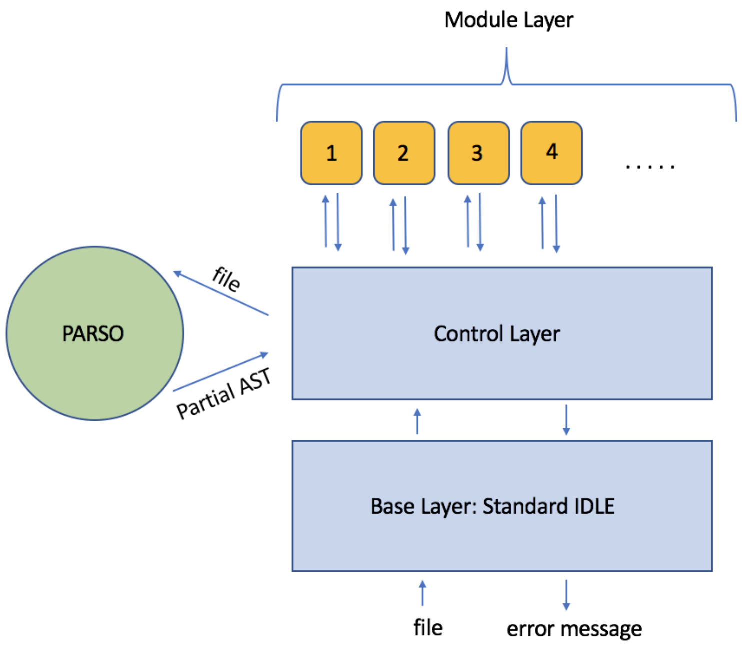


Figure 2-1: System Diagram

Parso

Parso is a parser that has the ability to construct partial syntax trees. It was built by David Halter and his collaborators as a part of an autocomplete and static-analysis tool for Python named Jedi [11]. They describe Parso as a "Python parser that supports error recovery and round-trip parsing for different Python versions ... also able to list multiple syntax errors in your python file."

Base Layer: Standard IDLE

Code enters the system through the base layer. The base layer has a copy of a standard Python tokenizer and parser, which all code is run through. In cases where the code does not contain a syntax error, no further analysis is performed and IDLE continues to run in the standard fashion. If the code contains a syntax error, this process fails and standard Python returns an error message. In these cases, the code is passed to the control level for processing.

Control Layer

Once code is passed to the control layer, a preliminary set of tests is run that checks for errors such as mismatched parentheses which can be found by tokenizing, with no need to parse the source. If such an error is identified, the control layer passes a corresponding message back to the base layer which displays it to the user. However, if no error is detected at the token level, the control layer passes the code to Parso and receives back a partial abstract syntax tree (AST). The control layer then sends this AST to the module layer. Each module checks the AST looking for a particular syntax error. The control layer continues sending the AST to modules until it either receives a classification for the error or reaches the end of the series of modules. At this point, the control layer passes an appropriate error message back to the base layer.

Module Layer

The module layer is made up a series of independent modules, each of which tests for a particular type of syntax error. The modules are ordered such that more specific tests run before tests that provide more general information. Each module looks at the context that surrounds an error in the partial AST to determine whether a certain mistake was made.

2.3 Implementation Details

This section elaborates on the details of the system and describes how they are implemented.

2.3.1 Parso

Parso has the flexibility to turn error-recovery either on or off. While error-recovery is off, Parso only builds a syntax tree when there are no syntax errors in the source. However, when error-recovery is on, Parso will construct a partial syntax tree even in the presence of errors. For our purposes, since we only run the code through Parso in cases where the built-in parser has already identified that there is invalid syntax, we always turn error-recovery on.

When a partial syntax tree is generated, Parso marks the token on which the parser recognizes that an error has occurred as an Error Leaf, although information about what type of token (name, operator, etc) it was originally is also preserved. The node suspected of containing the offending syntactical mistake is known as the Error Node.

2.3.2 Base Layer

One of the challenges of providing more in depth information about syntax errors in Python is the way in which Python tokenizes and parses code. Python's tokenizer returns a generator for the tokens in a piece of code and then iterates over the tokens. Unfortunately, if there is a syntax error in the code this iteration is interrupted by an error, all information about the previous tokens is lost and the program simply reports that the syntax is invalid. For our purposes, we modified this process so that each token is stored as it is generated so that in the case of a syntax error we have access to a partial stream of tokens leading up to the error. This partial stream of tokens is then passed on to the control layer along with the corresponding file.

2.3.3 Control Layer

The control layer has several separate functions, each of which is explained in a subsection below.

Expanding on Built-In Errors

In some cases, in the presence of syntax errors, Python itself is able to provide some information beyond the generic "SyntaxError: Invalid Syntax" error that we are attempting to improve. In these circumstances, we can apply a wrapper to make the message more understandable to a beginner without any additional work. We chose which of Python's built-in error messages to write a wrapper for based off of their prevalence in the MIT data set and potential improvement to the message. One such example is explored next.

Empty Block Example

Consider the following snippet of code where there is no block of code following the else command.

```
x = 4
if x == 4:
    print('cat')
else:
```

In standard IDLE, this results in a message of "Unexpected EOF While Parsing". To make this slightly more approachable to a beginner who might not be familiar with the acronym EOF, the improved error message instead reads "Unexpected End of File While Parsing".

Note that the improved error messages produced at this level are still not the most useful to students. In particular, in the example above the message still does nothing to suggest what might have caused the error. Further work could certainly improve upon these messages by performing more analysis to attempt to determine their root cause. However, since our tool was primarily focused upon improving on messages of "SyntaxError: Invalid Syntax" we do not perform any additional analysis at this level.

Token Level Errors

The next level of errors that the control layer handles are those errors that standard IDLE does not provide a more descriptive message for, but that can easily be detected by looking at the partial stream of tokens without any additional syntactical information. Errors processed at this level include mismatched brackets, parentheses, and curly braces.

For clarity, we will discuss errors with parentheses specifically, but tests for brackets and curly braces are analogous. To detect these errors, we scan through the stream and check that the number of open parentheses and close parentheses match. Note that the stream of tokens only spans up to where Python noticed that there was an error. In particular, this means that not every parenthesis in the file is necessarily in the stream. If at the end of the stream the number of open parentheses is greater than the number of close parentheses this indicates that a close parenthesis must be missing. Conversely, if there are more close parentheses than open parentheses an open parenthesis must be missing.

In our final tool, all information about mismatched parentheses, brackets and braces remains at this level.

Mismatched Parentheses Example

Consider the following code snippet which contains a syntax error because of mismatched parentheses:

```
a = 4
b = 2
c = 1
root1 = (-b + (b**2 - 4*a*c)**0.5/(2*a)
root2 = (-b - (b**2 - 4*a*c)**0.5/(2*a)
print(root1, root2)
```

The stream of tokens returned has 6 open parenthesis and 4 close parenthesis.

Since these are not matched when IDLE's tokenizer encounters an error, which in this case is the end of the file, we know that there must be a problem with mismatched parentheses. The error message returned in this case notes "Missing)" and highlights appropriately as shown above.

Misordered Parentheses Example

Another common situation to consider is when the file has a matching number of open and close parentheses but they are ordered in a syntactically incorrect way. Consider for example the following code snippet:

```
)))(((
```

In this case only a single token is created before an error is encountered. Therefore, the stream of tokens only has a single close parenthesis, so the error message "Missing (" is produced.

Partial Syntax Tree Level Errors

The cause of some errors cannot be determined from the token stream directly because the vast majority of syntax errors in Python are dependent upon the grammar specifications for particular types of statements and clauses. To classify these errors, the control layer needs a partial syntax tree in order to understand the structure of the code around the location of the error. As a result, if the control layer has not detected a more specific error message from the token level and built-in IDLE messages, it uses Parso to construct a partial AST.

Once Parso has constructed the partial syntax tree for the control layer, the next step is to determine the location of the first Error Leaf and Error Node in the tree, since this is the error that the tool will attempt to identify. These nodes are identified through a depth-first search.

The control layer then passes the partial syntax tree along with the first Error

Leaf and Error Node onto a series of modules in the module layer.

2.3.4 Module Layer

In keeping with our design goal of building a modular system, we built our classification mechanism as a series of separate module tests to check for different errors. The AST is run through the modules until one provides a more specific error message or they are all exhausted. Then, the control layer passes the information gleaned back to the user.

The tests typically combine several of the following elements:

- Look at the type or content of the Error Node
- Look at the type or content of the Error Leaf
- Check the beginning of the statement for a keyword
- Compare surrounding code to common patterns of errors (in particular patterns from other languages such as x++ or copying the terminal header directly into a file)
- Look at the type or contents of siblings of the Error Leaf and preceding and following leaves

Altogether, 31 separate tests for different specific syntax errors were constructed. Section A of the appendix details each new error message and the techniques employed within the module designed to identify it. In the following example we explain how one such representative module works.

Example Module Design

Consider the following snippet of code that contains a syntax error as a result of a missing colon at the end of a conditional statement.

```
x = 4
if x == 4
    print('cat')
```

This code sample is transformed into a partial syntax tree by Parso, which is displayed in figure 2-2.

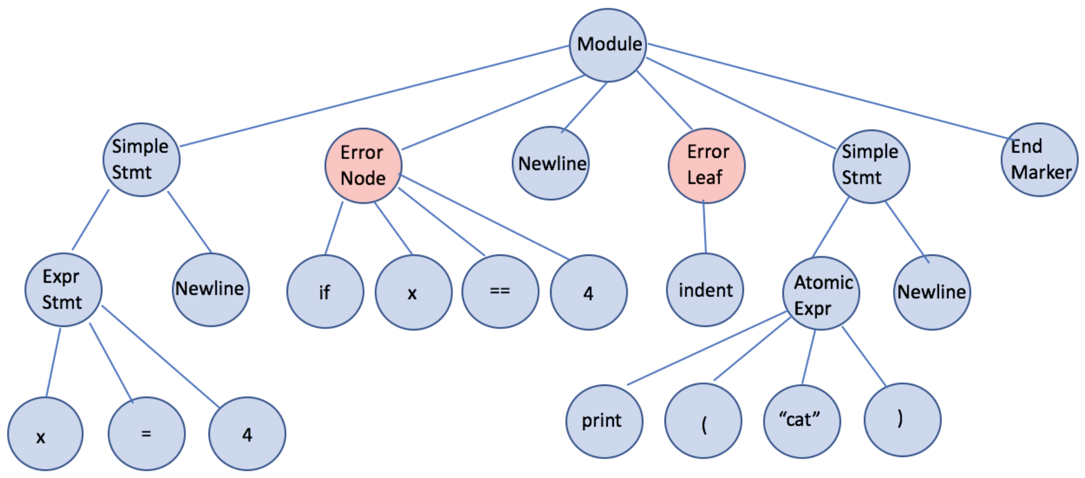


Figure 2-2: Partial Syntax Tree Generated By Parso

Since in this case there is only one Error Leaf and one Error Node in the tree, the nodes highlighted in red are the relevant Error Leaf and Error Node.

All of this information is passed to the module layer, and the module that identifies missing colons acts roughly as follows:

1. Start at the Error Node. Look through the leaves in order until you reach a newline. Keep track of whether or not you've found a colon.

2. Go to the Error Leaf. Check whether the Error Leaf was originally an indent and if the previous sibling, called s1, is a newline. If so, look at the last leaf of the sibling before s1, called s2, and check whether it ends in a colon.

3. If neither of the above found a colon, and s2 began with a keyword that requires the statement to end in a colon, report that a colon is missing from the statement.

In this case, since no colons were identified through steps 1 or 2 and the Error Node begins with the keyword `'if'` which requires a colon at the end, the module would return the more precise error message “`if` statements must end in a colon” along with highlighting as shown below.

```
x = 4
if x == 4
    print('cat')
```

2.4 Avoiding Excessive Errors

All of the modules we constructed to determine an appropriate error message are dependent on the local context around the Error Leaf in the partial syntax tree. While these tests provide valuable information when the code is close to syntactically correct Python, if the code is not sufficiently close to syntactically valid Python it is possible to trigger misleading or erroneous messages from our tool.

For example, consider the following code snippet from a student submission in the MIT data set:

```
if i != to m in range(10) where m="" and i!=".":
```

This single line contains a number of syntactical errors in Python. Unfortunately, the multitude of errors causes our system to reort an error message that reads “Misplaced or misused colon” since the student’s attempt at writing an `if` statement is not in the form of a statement that our tests recognize as needing to end in a colon. However, this message is certainly not ideal under these circumstances. Assuming the student intended to write a conditional statement, the colon at the end of the line is actually one element they did correctly and is not the key issue with the student code.

In keeping with our goal of not misleading students, we would like to avoid returning messages in situations where many consecutive errors might result in an incorrect or misleading error message. As a result, we implemented a check to see if code was reporting too many errors in a given line to return useful information about what should be done to fix the error. Empirically, lines containing more than three errors rarely produced useful error messages. Therefore in cases where four or more errors appear on the same line our tool simply states that there are too many errors to report additional information. This threshold can be easily adjusted.

2.5 Error Locations

A final element of the system is the machinery to determine the location where an error occurs. One advantage of our partial syntax tree based approach is that is allows us to provide a more exact error location based on the location of the error within the AST. To this end, the modules return not only an improved error message, but also tuples representing the beginning and end of the location that is the cause of the error.

Chapter 3

Extensions

Beyond the base system described in Chapter 2, we built a few extensions to the project. These extensions - namely the autocorrection of errors, ability to correct multiple errors at the same time, and an improved user interface - are described in this chapter.

3.1 Autocorrecting Errors

In the process of writing tests to distinguish between different types of errors, we realized that we could also use this fine-grained assessment of what exactly was going wrong in a particular piece of code and where the mistake was to attempt to automatically correct the syntax error. While this did not necessarily fit with our goal of designing a system to assist beginning coders in learning to understand and correct errors for themselves, we decided to implement it in order to help with testing how well our tool was working. Running code through the autocorrect process and then checking whether or not it is syntactically correct allows us to validate whether our classification of the original error was correct. Furthermore, correcting earlier errors in a file allows the tool to find and classify later errors within the same file. Note that our autocorrect feature only attempts to correct syntax errors and is not necessarily going to produce working code.

3.1.1 Autocorrection Process and Example

The general plan for autocorrect is to take advantage of the fact that in the course of determining what error message is appropriate in a given situation we have determined both where the error occurs in the code and what we think the student did wrong. To give an idea of how this plays out in practice, we explore the following example.

Consider the following partial syntax tree that results when trying to parse the following code snippet

```
x = 2
if x = 3:
    ....
```

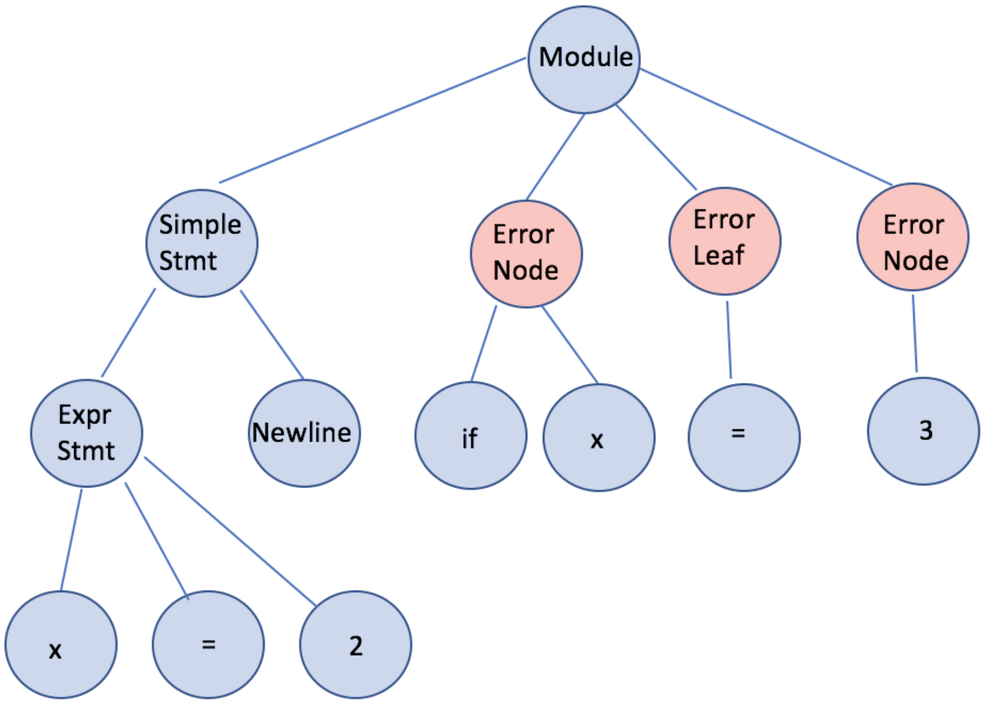


Figure 3-1: Original Partial Syntax Tree Generated By Parso

When we run the code snippet through our classification process we find that the module that detects use of = instead of == in a conditional statement is triggered.

Then, before returning the error message, we can take the Error Leaf from the parse tree and change the code that it represents, “=”, to the corrected code, “==”. The updated parse tree is shown in Figure 3-2.

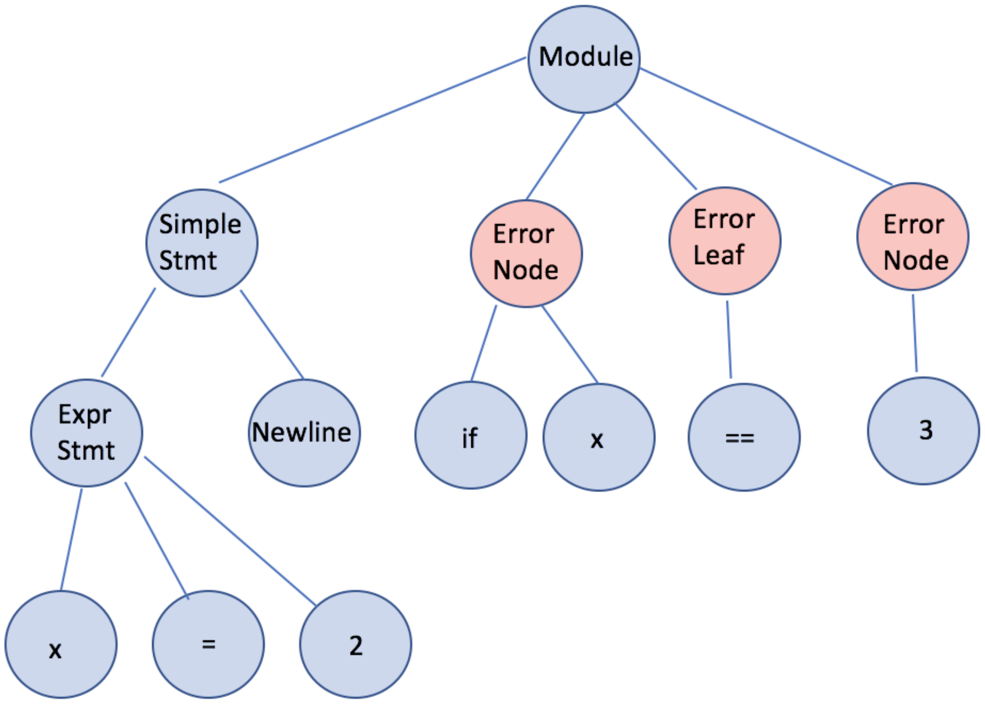


Figure 3-2: Partial Syntax Tree Generated By Parso after Correction

Note, however, that simply changing the code associated with the Error Leaf does not update the structure of the parse tree.

To accomplish this, we take advantage of the fact that Parso has a built-in function which returns a string corresponding to a given tree. In the above example, this gives us:

```
x = 2
if x == 3:
    ....
```

Next this new code is run through Parso again in order to create a new partial syntax tree. This new tree, as shown in figure 3-3 is now syntactically correct, which suggests that we correctly identified the syntactical issue in the original code.

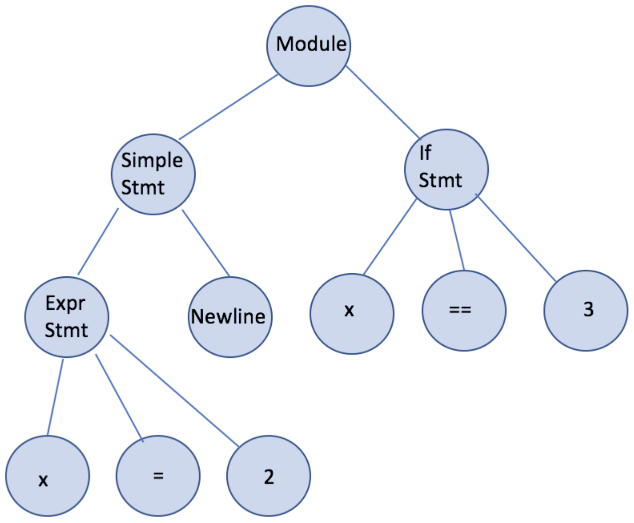


Figure 3-3: Partial Syntax Tree Generated By Parso after Reparsing

3.1.2 Autocorrect Implementation Details

One of the things that made implementing autocorrect particularly tricky is that it required modifying the underlying Parso code. In particular, Parso already had support for retrieving the string of the code associated with a given node, but it did not have support for reassigning a different string to replace the string of the original code. Fortunately, since the code associated with a node is simply an attribute of each node class, this change boiled down to adding a method to each class to allow the update. Unfortunately, however, the structure of the Parso code meant that this change had to be made in multiple locations since the location we would like to edit is not always the Error Leaf, but could be a child of the Error Node or another node entirely.

Another challenge was that a lot of errors are much easier to identify than they are to correct. For example, one of the tests looks for misuse of curly braces around blocks of code. This is a common mistake pattern for students who are learning Python for the first time, but already have experience in another language, such as Java, that uses curly braces to delineate blocks of code. It is not too difficult to identify when these braces have been misused in Python, but it is much more difficult to correct this type of error. There are a wide variety of potential structures that students might have been attempting to build and no one fix that would suit all of them. As a result, autocorrection was not implemented for all the errors that we classified.

3.2 Dealing with Multiple Errors

Currently standard Python is only able to report a single error within a file even if multiple exist. In some cases, seeing all the syntax errors in a piece of code at the same time might be ideal because in many cases multiple errors are related and it is easier to fix them by observing all the error messages at the same time. Also, observing multiple errors at the same time makes it easier for us to automate checking whether or not we have been able to correct the errors in the code by reducing the number of times that we have to iterate the above autocorrection process.

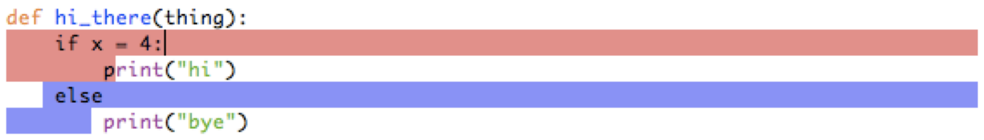
One of the neat features of Parso that we can exploit is the fact that it builds partial syntax trees that can contain multiple Error Leaf nodes. As a result, Parso can list all of the Error Leaf nodes within the tree to see multiple potential errors at once. A challenge, however, is that the method we relied on previously to find the first Error Leaf and the corresponding Error Node has to be adapted to correctly pair as many sets of Error Nodes and Error Leaf nodes within the tree as possible.

In our work on the single error case we observed that in the majority of cases the corresponding Error Node is the previous sibling of the Error Leaf. As a shortcut in the multiple error case, we guess that this is true and for each Error Leaf set the corresponding Error Node to be the previous sibling. Then, we run each pair through the module layer in the same way as described in Chapter 3. We collect all results, and return a list of more descriptive error information for each pair if we are able to classify it.

Unfortunately, the algorithm we used to match multiple Error Node and Error Leaf pairs is not as complete as the solution that only attempts to match the first pair, particularly because in some cases our guess for the Error Node is incorrect. In cases where the guess is incorrect, none of the modules introduce incorrect error information because each module is explicitly looking for a syntax violation. Instead, in these cases we simply return that we have no additional information to add. As a result, there are some cases where attempting to match multiple pairs is altogether unsuccessful since none of the guesses of corresponding Error Nodes are correct. In these cases, we default to only trying to match the first pair and proceed as if in the single error case.

3.2.1 User Interface

The final extension to the tool was a more elaborate user interface than IDLE typically employs. This entire portion of the project was developed by Samantha Briasco-Stewart, a collaborator, but we will quickly summarize the elements of this extension.

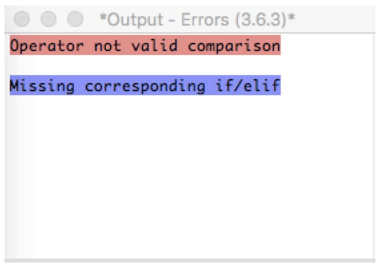


```
def hi_there(thing):
    if x = 4:
        print("hi")
    else
        print("bye")
```

Figure 3-4: User Interface Highlighting

While standard IDLE has built-in highlighting already, we adjusted it so that our new, more correct error locations would be the new locations of the highlighting. We also implemented highlighting for multiple errors where each error is highlighted in a unique color. These changes are reflected in figure 3-4.

Another update was to have the error messages appear in another window instead of in a modal dialog. An advantage of this approach is that, while the dialog has to be closed before the changes can be made, the new window allows the user to reference the error messages while they are working on correcting their code. As shown in figure 3-5, this new error window shows the error messages with highlighting corresponding to the location of the error in figure 3-4.



```
Operator not valid comparison

Missing corresponding if/elif
```

Figure 3-5: User Interface Error Window

Chapter 4

Results

The first step to providing more helpful and understandable syntax error messages for beginning coders is to properly classify the syntax errors. To this end, we used the historical data set from MIT introductory courses to test how well our tool was able to classify the errors we encountered.

4.1 Hand-Checked Results for Training Set

During the development of the tool we used a data set drawn from 6.145 submissions to guide which particular syntax errors we addressed and check that tests were working as intended. This training set consisted of 1048 examples. In this data set, the modules cover 87.5% of the submissions with an error message that is more specific than "invalid syntax". Of the examples classified, 99.3 % were classified correctly when we checked the error messages by hand. This data is discussed in Section 5.1. Table 4-1 contains the raw data results.

Table 4.1: Training Data Set Results

	Total	Unclassified	Classified Correctly	Classified Incorrectly
Raw Number	1048	131	911	6
Percentage	100 %	12.5 %	86.9 %	0.6%

Table 4.2: Testing Data Set Results

	Total	Unclassified	Classified Correctly	Classified Incorrectly
Raw Number	1048	162	886	19
Percentage	100 %	15.5 %	82.7 %	1.8%

4.2 Hand-Checked Results for Test Set

Once the construction of the tool was completed, we used another data set from 6.01 Spring 2016 to test the performance of the tool on previously unseen examples. Instead of pulling submissions sequentially from the database, we ensured that only one submission from any single student on a given assignment was included in the data set. For this analysis, we again only attempted to correct a single error and did not use autocorrect.

The test data set again consisted of 1048 examples. Of these snippets, 84.5% were matched with a more specific error message than "invalid syntax" when run through the tool. Of the examples classified, 97.9 % had an error message that was correct when hand-checked. This data is discussed in Section 5.2. Table 4-2 contains the raw data results.

4.3 Coverage Results

Additionally, we ran coverage tests on a larger set of data. The examples for this analysis were pulled from 6.01 Spring 2016 submissions. In this case, we only compared those the tool classified versus those that it did not; the classifications were not checked for correctness. The raw data is listed table 4-3.

Table 4.3: Coverage Results

	Total	Unclassified	Classified
Raw Number	5000	616	4384
Percentage	100 %	12.3 %	87.7 %

Table 4.4: Autocorrect Results

	Total	Classified	Fixed
Raw Number	1048	859	285

4.4 Autocorrect Results

We also ran example submissions with autocorrect turned on to test how many of the syntax errors autocorrect seems to be able to correct. For this analysis, examples were only run through the classification and autocorrect process a single time and then checked to see whether the resulting code was free of syntax errors. In particular, this means that for any examples that originally contained more than one syntax error we only attempted to correct the first error, so the corrected file is guaranteed to still contain other errors. Despite this limitation, about a third of the submissions were syntactically correct after the autocorrect process. The raw data is available in table 4-4.

Chapter 5

Discussion and Future Work

This chapter discusses the raw results presented in Chapter 4 and then explores directions for future work.

5.1 Hand-Checked Results for Training Set

Perhaps unsurprisingly, since the training data was used to help decide which modules should be developed, the tool effectively addresses the types of syntax errors encountered in the training data and covers 87.5%.

However, more importantly to the main design goal of accuracy, of error messages classified 99.7 % were correct when hand-checked. Unfortunately, there are still a few misleading or inaccurate error messages produced. Among the few examples in the training set that still get incorrect classifications are

```
if: m>=1 and m<=2:
    ....
```

for which the tool produces an error message of "`if` requires a condition" and

```
new_word==
```

which results in the tool producing "incomplete variable assignment". Both of these erroneous classifications could potentially be amended by re-wording the text of the error message used in each situation. For example, the first could instead report "Malformed `if` Statement" and the second could suggest "Incomplete Assignment or Comparison". To avoid the lose of specificity that this would incur, another option would be to add additional modules to address those specific errors.

5.2 Hand-Checked Results for Test Set

Performing the same analysis on the test set further indicates that for the vast majority of classifiable code samples the tool produces a correct error message since 97.8% of snippets classified were correct. However, the set of incorrect classifications is larger in the test set than the training set. This could be a result of the tool being developed to better target the errors observed in the training set than those present in the test set.

In particular, on the test set the tool seemed to struggle to correctly classify errors related to the return statement within a function. One of the commonly misclassified error patterns was:

```
def f():
    return DDist({ideal[state] = 1})
```

This was assigned an error message of "Operator not valid comparison" because of the use of the equals sign. Essentially similar errors accounted for four of the nineteen misclassifications. Another common error pattern was of the form

```
def f():
    return=Gain(k)
```

which also raised an error message of "Operator not valid comparison" and again accounted for another four of the nineteen misclassifications. Finally, the bulk of the errors were of the form

```
def f():
    x = return 5
```

which received an error message of “Incomplete Variable Assignment” and accounted for seven of the nineteen errors.

While all of these error message are incorrect and misleading, the fact that the majority of the errors center around malformed return statements suggests that a specific module that checks for syntactically incorrect return statements could be added near the beginning of the series of modules and potentially avoid these misclassifications altogether.

In terms of coverage, the test set also performed worse then the training set. This is also to be expected because modules were built to represent the errors present in the training set. However, together the coverage and accuracy results on the test set are very encouraging because they suggest that the modules built work appropriately and in the future more modules could likely be built to provide greater coverage of a wider variety of syntax errors.

5.3 Coverage

Mirroring the above analysis, the results from the larger coverage study show the modules cover 87.7 % of the syntax errors in a larger sample of the 6.01 submissions. This result is important because it is based off a larger set of novice code samples than were considered for the hand checked results above and confirms that the modules are effectively covering the errors that novices are encountering in MIT courses.

Table 5.1: Most Common Error Messages in Coverage Data Set

Error Message	Count
Missing)	1619
Statement must end with a colon	551
Missing (	484
Operator not valid comparison	297
Two variable names in a row	241
Missing parentheses in call to “print”	237

The distribution of error messages in the coverage data set supports previous research showing that a small number of unique errors make up the bulk of the errors that novice coders encounter. In particular table 5.1 shows the most common error messages that were assigned along with their frequencies. Together these six error messages account for almost 80% of the classified errors. Note that besides the last item in the table, all of these categories of errors result in “SyntaxError: Invalid Syntax” when run through standard Python. This suggests that the tool is providing additional insight on errors that students are commonly making, as desired.

The set of code snippets that remain unclassified during the coverage test also provide valuable information about what sorts of syntax errors our tool does not have modules to classify. Sorting through some of these unclassified examples by hand suggests syntax errors that future modules might want to address. A few of those observed include:

- Incomplete operations with operators other than +, * , - , or / .
- Incorrect syntax for dictionaries. We observed some submissions where not every key in a dictionary was assigned a corresponding value. Another mistake was attempting to access elements of a dictionary by writing `dict{x}` instead of `dict[x]`
- Malformed lambda functions. In particular, some submissions attempted to pass a tuple as a parameter.

Table 5.2: Most Common Error Messages in Autocorrect Data Set

Error Message	Count
Statement must end with a colon	124
Operator not valid comparison	77
Two variable names in a row	18

5.4 Autocorrect

Finally, the autocorrect results are also encouraging. Since the autocorrect extension currently only corrects a single error, even if it performed perfectly it still would only be able to totally eliminate errors in those examples that have a single syntax error. Combined with the fact that not all modules have autocorrect implemented, the 33% success rate is outstanding. If nothing else, these results serve as a proof of concept that autocorrect could be further developed and run with multiple iterations to address a significant portion of errors that students make.

Furthermore, the limited pool of errors for which autocorrect was implemented further supports the idea that a small number of errors account for a large portion of the mistakes made by novice coders. Of the code snippets that autocorrect was able to fix in this test, table 5-2 displays the most frequent original error messages. Note that the error messages in table 5-2 correspond to the most frequent error messages observed in the coverage data set with the exception of those indicating a missing parenthesis. For the missing parenthesis cases, autocorrect was not implemented. Altogether, there were twenty-one distinct error messages where autocorrect was able to, in at least one case, successfully correct the error.

5.5 Future Work

One of the natural next steps in developing the tool would be to collect and analyze data as to its effectiveness and usability. To this end, it would be possible to run an experiment and compare beginning Python programmers' debugging performance utilizing our tool and a standard version of IDLE.

There is also room for improvement within the tool without access to additional data. Improvements on the current components of the tool might include:

- Identifying Error Nodes for Multiple Errors - The current algorithm for matching Error Leafs with Error Nodes could be improved by searching for the previous Error Node instead of guessing that it is the previous sibling node.

- Autocorrect Predictions - The existing autocorrection algorithm is designed simply to produce syntactic correctness. For example, it always predicts * for missing operations. Creating a more intelligent prediction scheme for many of the corrections would improve the usability of the tool.

- Additional Modules - Not all syntax errors observed in the MIT data set are covered by modules. Additional modules could be added to increase coverage. In particular, the issues with return statements encountered in the test set could be addressed.

Beyond improvements to existing features of the tool, future extensions of the project might include:

- Style Help - Often novice coders students learning to code write code in ways that are functional but inefficient and redundant, such as typing "x=x+1" instead of "x+=1". It would be interesting to create a tool to offer such suggestions to students to help aid their style development and increase their efficiency.

- Integration with CAT-SOOP - Many MIT courses have students submit their exercises through the CAT-SOOP online tutor [11]. Integrating our tool into CAT-SOOP would make it easier for MIT courses to utilize the tool.

5.6 Conclusion

Many novice Python coders have difficulty debugging their code since standard Python error messages are geared towards experienced users who are familiar with programming principles. The tool described in this thesis is designed to provide more specific and understandable error messages to help beginners understand and fix their errors. In simple cases, the tool either elaborates on built-in Python error messages or analyzes the code at a token level without the need to parse the code. However, in most cases, the tool needs more syntactical information and classifies errors by constructing a partial syntax tree, identifying the error within the tree, and examining the surrounding elements of the code. In a study of 1048 submissions from an introductory MIT Python course, the tool assigned a classification to 84.5% of the code samples. Of this classified portion, 97.8% were correctly classified. This suggests that the tool is able to accurately classify student errors and does a respectable job of covering the syntax errors encountered by novice coders in introductory MIT courses.

Appendix A

Module Error List

The following list contains the new error messages determined at the module level along with a brief description of how each module works and an example of a code snippet that would trigger the respective error.

1. "Invalid python. Likely copy and pasted header from the terminal" - The nodes surrounding the source of the error are the text from a standard terminal header. For example,

```
     Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
   information.
>>> x = 4
```

2. "++ is not a valid operator" - The Error Node ends with "++". For example,

```
x = 0
for x in range(12):
   x++
```

3. "- - is not a valid operator" - The Error Node ends with "- -". For example,

```
x = 0
for x in range(12):
   x--
```

4. “ { } for dictionaries only” - The line begins with a keyword for a conditional and is followed by a colon and then { } to contain the indented block. For example,

```
x = 0
if x == 2:
   x = 1
else:{
   x = 0
    }
```

5. “Function definition needs ()” - The line begins with the keyword `def` and the Error Leaf is on a colon. For example,

```
def f:
   return 3
```

6. “Function definition must have a name” - The line begins with the keyword `def` and the Error Leaf is on an open parenthesis. For example,

```
def ():
   return 3
```

7. “Function arguments must be variables” - The line begins with the keyword `def` and the node before the Error Leaf is a number. For example,

```
def f(4):
   return 3
```

8. “The keyword ‘pass’ should not be followed by anything on the same line” - `Pass` is not followed by a new line character. For example,

```
def f(x):
   pass x
```

9. “Missing corresponding ‘try’ ” - Line begins with `except`, but the previous block doesn’t begin with `try`. For example:

```
def f(x):
    except ValueError:
        x = 3
```

10. “Can’t assign value to variable in return statement” - The Error is on the equal sign in a return statement and the previous node is a variable name. For example,

```
def f(x):
    return x = 4
```

11. “Can’t assign value to variable in return statement; comparison with invalid comparator ” - The error is on the equal sign in a return statement and the previous node is not a variable name. For example,

```
def f(x):
    myList = [2,3,4]
    return [1,2,3] = myList
```

12. “Incomplete variable assignment” - The Error Node ends in an equals sign and previous to it is a variable name. For example,

```
x =
```

13. “Variable types do not need to be declared” - The Error Node represents the name of a variable type and the next node is another variable name. For example,

```
int x
```

14. “Operator not valid comparison” - The error occurs in a statement that begins with a key word that requires a comparison and the error is on an equals sign. For example,

```
x = 6
if x = 5:
    x = 7
```

15. "`if/else/elif` statement must end with a colon" - The line begins with a conditional and ends without a colon so the Error Node is on a newline. For example,

```
x = 6
if x == 5
   x = 7
```

16. "Incomplete operation" - There is an operator (+, -, *, /) not followed by a number. For example,

```
x = 3
y = x +
```

17. "Two variable names in a row" - The error occurs on a variable name and the preceding node is also a variable name. For example,

```
my int = 5
```

18. "Missing operator; need * for multiplication" - The Error Node is a number which is followed by a variable name. For example,

```
a = 3
x = 2a
```

19. "Missing operator between numbers" - The Error Node is a number which is followed by another number. For example,

```
a = 3 + 5 2
```

20. "Malformed `for` loop" - The line begins the keyword `for`, but is not syntactically valid. For example,

```
x = 3
for (i in range(3)):
   x += 1
```

21. "Malformed `'for'` loop; missing `'in'` in `for` loop" - The line begins with the keyword `for`, but is missing the `in` before range. For example,

```
x = 3
for i range(3):
   x += 1
```

22. "Malformed `'for'` loop; uses comparison instead of iteration" - The line begins with the keyword `for` but contains a comparison. For example,

```
x = 3
for i < 3:
   x += 1
```

23. "Missing corresponding `if/elif`" - The line begins with the keyword `elif` or `else`, but the previous block does not begin with `if` or `elif`. For example,

```
x = 3
if x == 4:
   x = 5
y = 7
else:
   x = 6
```

24. "Should be `elif` instead of `else if`" - The Error Node ends with the keyword `else` and is followed by the keyword `if`. For example,

```
x = 3
if x == 4:
   x = 5
else if x == 7:
   x = 6
```

25. "Comparison operations must have at least two operands" - The line begins with a keyword that requires a comparison and there is an `and/or` but the Error Node falls on a comparison operator. For example,

```
x = 3
if x < 4 or >6:
   x = 5
```

26. "! is not an operator in Python" - The Error Leaf falls on a '!'. For example,

```
x = True
if !x:
   x = 5
```

27. "Misplaced or misused colon" - The Error Leaf is a colon, but the line does not begin with a keyword that requies a colon. For example,

```
x = True
if x:
   x = 5:
```

28. "Malformed string; missing ' " - The Error Leaf either begins or ends with a ' but is missing the other. For example,

```
x = ['a','b','c',d','e','f']
```

29. "'`elif`' requires a condition" - The line begins with `elif` but is missing a condition before the colon. For example,

```
x = 5
if x == 6:
   y = 7
elif:
   y = 8
```

30. "'`if`' requires a condition" - The line begins with `if` but is missing a condition before the colon. For example,

```
x = 5
if:
   y = 7
```

31. "'`else`' should not have a condition" - The line begins with `else` and is followed by anything other than a colon. For example,

```
x = 5
if x == 6:
   y = 7
else x == 7:
   y = 8
```

Bibliography

[1] Pritchard, David. *Frequency Distribution of Error Messages.* Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools - PLATEAU 2015 (2015): n. pag. Web.

[2] Jackson, J., M. Cobb, and C. Carver. *Identifying Top Java Errors for Novice Programmers.* INFONA. Frontiers in Education Conference, 19 Oct. 2005. Web. 05 May 2017.

[3] Nienaltowski, M. Pedroni, M. Meyer, B. *Compiler Error Messages: What Can Help Novices?* ACM SIGCSE Bulletin 40.1 (2008): 168. Web.

[4] Guo, Philip. *Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities.* ACM. N.p., 07 July 2014. Web. 05 May 2017.

[5] Hristova, Maria, Ananya Misra, Megan Rutter, and Rebecca Mercuri. *Identifying and Correcting Java Programming Errors for Introductory Computer Science Students.* Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education - SIGCSE '03 (2003): n. pag. Web.

[6] Marzieh Ahmadzadeh, Dave Elliman and Colin Higgins (2007) *The Impact of Improving Debugging Skill on Programming Ability*, Innovation in Teaching and Learning in Information and Computer Sciences, 6:4, 72-87

[7] Toomey, Warren. *Quantifying The Incidence of Novice Programmers Errors*

[8] Flowers, T., C.a. Carver, and J. Jackson. *Empowering Students and Building Confidence in Novice Programmers through Gauntlet.* 34th Annual Frontiers in Education, 2004. FIE 2004. (n.d.): n. pag. Web.

[9] Denny, Paul, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. *Understanding the Syntax Barrier for Novices.* Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education - ITiCSE '11 (2011): n. pag. Web.

[10] Denny, Paul, Andrew Luxton-Reilly, and Dave Carpenter. *Enhancing Syntax Error Messages Appears Ineffectual.* Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education - ITiCSE '14 (2014): n. pag. Web.

[11] Hartz, Adam. *CAT-SOOP : a tool for automatic collection and assessment of homework exercises* Masters thesis, Massachusetts Institute of Technology, June 2012.

[12] Halter, David. *Parso - A Python Parser* https://github.com/davidhalter/parso.