Making Python Easier to Learn with Improved Syntax Error Reporting

by

Samantha Briasco-Stewart

S.B., Massachusetts Institute of Technology (2017)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

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

Certified by...... Signature redacted ....
Adam Hartz
Lecturer
Thesis Supervisor

Signature redacted
Accepted by ....
Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

Making Python Easier to Learn with Improved Syntax Error Reporting

by

Samantha Briasco-Stewart

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 Electrical Engineering and Computer Science

Abstract

In this thesis, we examine which types of compilation-time errors are most prevalent among Python learners at MIT. Then, in order to improve Python's error reporting, we design and implement a system to describe more accurately selected syntax errors, the least well described of the prevalent errors. This system is tested automatically against a set of hand-classified syntax error-producing code samples, as well as by hand against an unclassified set. Lastly, we design and implement a graphical interface to support the description system, and integrate this graphical interface into a popular Python development environment, *IDLE*.

Thesis Supervisor: Adam Hartz
Title: Lecturer

Acknowledgments

First, I would like to thank Adam Hartz, my thesis advisor, without whom this project would have been over before it was started. He is an amazing teacher, and I am glad to have had the chance to work with him.

Next, Anne Kelley, my partner, for her work on her half of this project. I could not have asked for a better person to work with.

I would also like to thank Kat Hendrickson for struggling together with me, Graeme Campbell for forcing me to write and saying it was okay when I couldn't, and Kade Phillips for listening to my ideas, even when they were terrible.

Lastly, I would like to thank my family, for their love and support, and for reading my drafts and editing them with the keenest of eyes.

Contents

List of Figures

Chapter 1

Introduction

According to Lahtinen et al., one of the most difficult problems novice programmers encounter is responding to errors in their code [9]. The hypothesis leading to this thesis' topic is that this difficulty stems from three sources:

1. inability to understand the error message
2. inability to find the source of the error
3. inability to fix the error

Novices are often unable to understand why a computer fails to execute correctly the code they write, as they assume the computer possesses some degree of intelligence and can therefore reason about code in the same way a human might [13]. Because of this failure to understand why the computer is unable to execute a particular piece of code, the novice then is unable to figure out *how* to find the cause of the error, and is thus unable to fix it. For more seasoned programmers, this is less of an issue—seasoned programmers have a better understanding of the way a computer works, and using this understanding they can read more meaning into a received error message, enabling them to locate the cause quicker, and then fix the cause more easily [12].

These difficulties are exacerbated by error messages that are poorly-constructed and difficult for novices to understand [11]. Error messages often use technical jargon to describe the errors that occur, and in many cases do not provide much to indicate the source of the errors. To correct this, we have developed a system to provide better-constructed error messages that are more helpful for novice programmers. In particular, and for reasons discussed in the following sections, we chose to focus specifically on syntax errors in Python.

1.1 Python Errors

```
1 def squared(x):
2 # square x
3     squared_x = x * z
4 # return the square of x
5     return squared_x
6
7 print(squared(2))
```

Figure 1-1: A simple Python program

In Python, some error messages give more useful information than others. As an example, see Figure 1-1. This Python function has one error: on line 3, the programmer uses a variable *z* that has not yet been defined. The error message produced when this code is run is as follows:

```
> python ex1.py
Traceback (most recent call last):
  File "ex1.py", line 3, in squared
    squared_x = x * z
NameError: name 'z' is not defined
```

Figure 1-2: A helpful Python error message

This error message is helpful: it describes where the error is (line 3), in what function the error is ('squared'), and what the error is (a variable is not defined). The error message also gives the specific variable that is not defined (*z*) and prints out the line where it is first referenced. This is a well-constructed error message—it explains what the error is, using relatively simple language, and emphasizes the actual source of the error, in line with the guidelines put forth by Marceau et al [11].

(a)

```
def times(x, y):
    return = x * y
```

(b)

```
def times(x,y:
    return x * y
```

(c)

```
def times(x, y):
    retrun x * y
```

(d)

```
def times(x, y):
    return result = x * y
```

(e)

```
def times(x, y):
    return x y
```

(f)

```
def times(x, y):
    pass x * y
```

Figure 1-3: Some syntax errors

However, not all of Python's error messages are this helpful. Consider the examples in Figure 1-3. These are six versions of a function intended to return the product of two numbers, `x` and `y`, each with a unique error. Each, when run, produces the error message "`SyntaxError: invalid syntax`", despite each function's error having a distinct cause. In **(a)**, the user tried to set `return` (a keyword) equal to something (`x*y`). In **(b)**, the user forgot the closing parenthesis in the function definition. In **(c)**, `return` was misspelled as `retrun`. In **(d)**, the user tried to return a variable declaration. In **(e)**, there is no `*` between `x` and `y`, and in **(f)**, the user tried to use `pass` instead of `return`.

Unlike the message given with the `NameError` from Figure 1-2, the `SyntaxError` message returned from the examples in Figure 1-3 provides barely any useful information. The message describes the error as being caused by "invalid syntax", but without knowing what 'syntax' is or what would make it 'invalid', there is no information here to help a programmer determine the cause of the error, whether it is a missing parenthesis, a misspelled keyword, or an oddly-placed equals sign.

1.2 Past Work

Previous efforts to assist students with finding and fixing errors in code have focused on studying three main areas:

1. Which errors are most prominent, and students' opinions about finding and fixing errors

2. How to display error messages such that they effectively assist students in finding and fixing errors

3. How to help students fix errors, once they are found

We will discuss the efforts in each area separately, in the following sections.

1.2.1 Cataloging Errors

Finding and fixing programming errors is a significant (and important) part of learning programming, according to Lahtinen et al. in a 2005 study [9]. The authors found that novices, when asked, stated that finding and fixing bugs in their own programs was the most difficult part of learning to code [9]. One source of these bugs is that some novices assume the computer is smarter than it actually is. They believe that it has knowledge of the future and can read ahead in its own code, or that it can execute multiple lines at once, or that it can keep one condition in mind while executing another line of code [13]. Another common source of errors is typographical mistakes, compounded with misunderstandings of syntax.

We analyzed a dataset consisting of all submissions made to the online submission website for 6.01 and 6.S080 (introductory MIT EECS courses) in the past year (Figure 1-4). Paying attention only to the type of error, the most common errors in the dataset were:

1. `NameError`, 22.9%

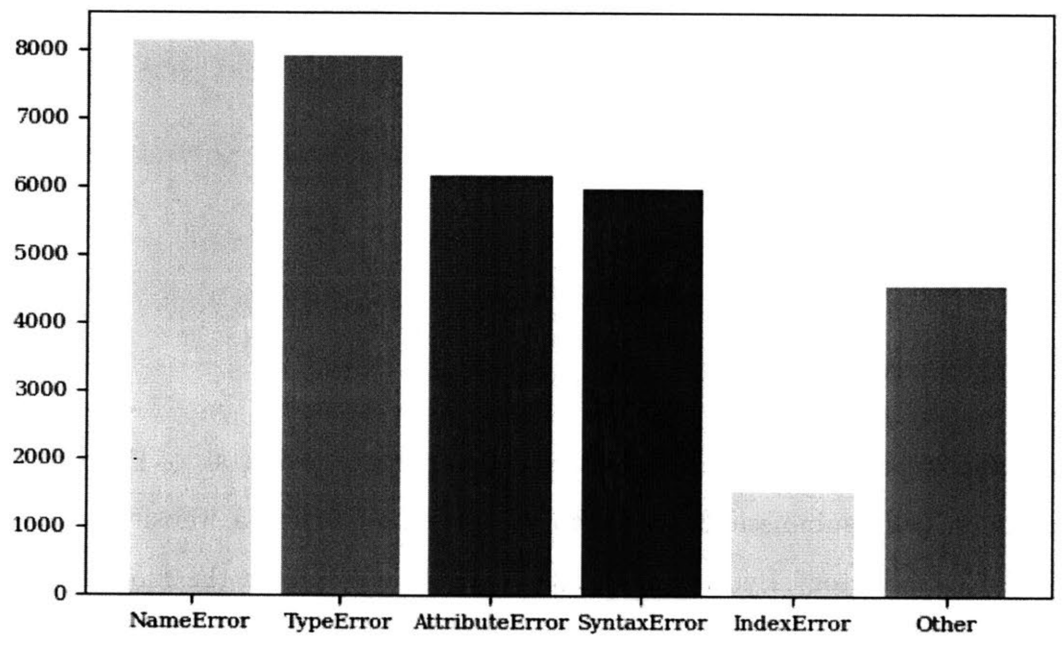
2. `TypeError`, 22.2%

3. `AttributeError`, 17.3%

4. `SyntaxError`, 16.8%

In comparison to this dataset, a 2015 study by Pritchard et al. of Python errors occurring in code submissions to *Computer Science Circles* (a website with short courses in introductory Python) found that syntax errors were the most frequently occurring error (as compared to fourth most frequent), occurring almost twice as often (28.1% to 15.2%) as the second most frequently occurring error [14].

There are a few possible explanations for this discrepancy. One is that 6.01 and 6.S080 encourage students to use their own text editors (eg. *IDLE*) to write code, submitting it to the website once it is ready to be graded. In contrast, the website used in Pritchard's 2015 study encouraged students to write code directly in the online checker, giving them a higher percentage of errors overall—640,000 out of 1.6 million (40%), compared to the MIT dataset's 35,659 out of 217,019 (16.4%). It is possible that because some students at MIT edit their code in a different editor and run it before submitting, they catch syntax errors that otherwise would have been counted, biasing the MIT dataset compared to Pritchard's. Another possible explanation, perhaps less likely, is that MIT's submission website (and the students' own editors) use Python-oriented syntax highlighting, which can help students recognize some types of syntax errors (eg. misspelled keywords) more easily. Pritchard's study's website does not have syntax highlighting, possibly leading to more spelling-based syntax errors, or just more syntax errors in general.



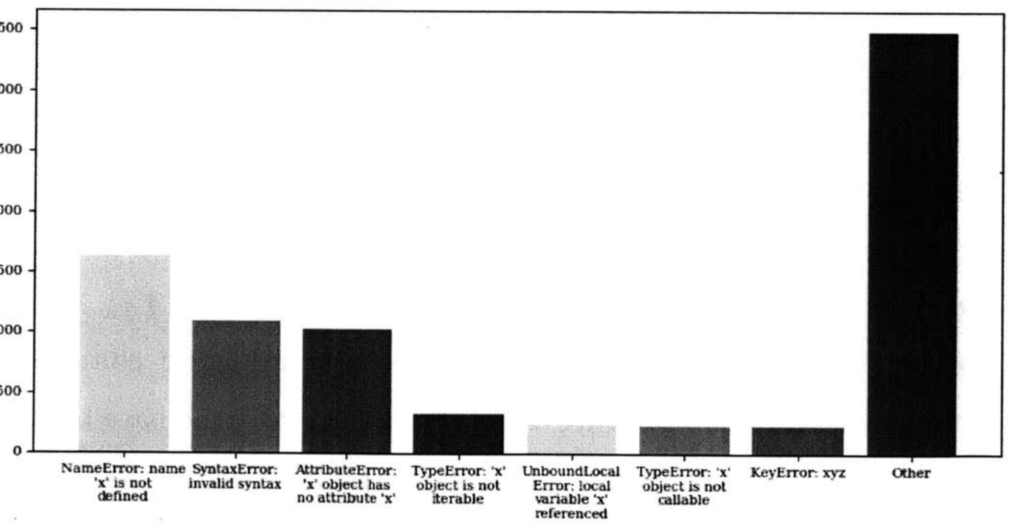


Figure 1-4: Error type (top) and message (bottom) frequencies in code submissions to 6.01 and 6.S080 at MIT

In the MIT dataset, if we include the message attached to the error, rather than just looking at the type, syntax errors rise to third place (Figure 1-4, bottom):

1. `NameError: name 'x' is not defined`, 22.9%
2. `SyntaxError: invalid syntax`, 15.3%
3. `AttributeError: 'x' object has no attribute 'x'`, 14.3%

These error messages were filtered to replace variable and type names with generic defaults, so that the message categories would not be too specific to give meaningful results. Syntax errors are specifically interesting in this analysis, as they required no filtering at all to reach third place, unlike name and attribute errors.

1.2.2 Displaying Errors

Another focus is on studying how to display error messages to students such that they are able to find and fix the causes of those error messages effectively. Interestingly, the results of studies in this area are sometimes contradictory. In a paper on compiler error messages, Nienaltowski et al [12] examined the effects of various types of error messages—long explanations, more terse versions, and entirely visual messages (highlights, etc)—on the accuracy and speed of computer science students who saw and attempted to correct them. The results of this examination were interesting: the authors found no significant relationships between the kind of error message and the accuracy or speed of the correction [12]. This was not supported by other papers, however; an earlier paper by Marceau et al. [11] concluded that error messages with shorter words were easier for novices to understand, and visual error messages actually helped users, as long as the visualization emphasized the actual source of the error in the code [11].

Lee and Ko wrote an application they called *Gidget* as another method of displaying errors. They intended this application to personify the computer, so that novice programmers could blame the computer rather than themselves, when their code didn't work [10]. By framing error messages in such a way as to make the computer seem apologetic when unable to execute a program, Lee and Ko found that users were less discouraged when encountering errors, viewing the error as the computer's fault for not understanding, rather than the user's fault for writing bad code. They observed that, when presented as a game, novices using *Gidget* completed median 5 levels, while novices with a similar (but not personified) editor completed only median 2 levels [10].

Another application, *Detective*, written by Hartz [6], focuses on giving helpful hints to users faced with certain types of Python errors. Users faced with a name error, for example, will see a list of variables from their code that are similar to the variable the error is complaining about, on the chance that they misspelled one of those [6].

1.2.3 Fixing Errors

The final focus of past efforts is on studying how to help students effectively fix errors once they are found. One application, *Whyline*, written by Ko and Meyers [8], enables users to run their own (Java-based) graphical applications inside the application. Then, after executing a series of clicks, the user can ask questions to *Whyline* about why their application did or did not do a certain operation [8]. The authors found that, compared to a previous study of experts debugging Java code without the help of *Whyline*, users of the system (ranging from novices to experts) were able to complete debugging tasks in less than half the time. (On average, users of *Whyline* took 4 minutes to complete a task, as compared to the average of 10 minutes from the previous study).

Another paper, by Hartmann et al., describes a system that crowd-sources solutions to error messages in Java and C++, so that students who have found an error can easily see several possible solutions, rather than having to brainstorm them on their own [5]. Hartmann et al. found that this approach worked fairly well, with on average about half of the queries to their system resulting in suggestions that students found useful.

1.3 Summary

The problem that we set out to solve in this thesis, then, is that Python's syntax error messages are uniquely unhelpful when compared to Python's error reporting in general, especially when the programmers encountering the error messages are novices. Past research confirms that novices experience significant difficulty in debugging syntax problems in Python. Other efforts concluded that simplifying error messages assisted novices in more quickly and easily finding and fixing bugs in their programs, although those efforts focused on different languages. (Marceau, for example, focused on *DrScheme* [11].) To solve the problem, we wrote a two-part system that first classifies syntax errors into smaller, well-defined categories, and then labels each with a distinct, helpful message and error location. We also integrated this system into *IDLE*, a popular Python editor.

Chapter 2 describes the inner workings of our system. This includes: a script we used to manually classify several hundred example faulty code segments; the parser we used to re-parse the faulty code and the heuristics we used to define each category of error; scripts we used to manually and automatically test the workings of the parser/classifier; and the integration of the parser/classifier into *IDLE* and additional features added thereafter. Chapter 3 contains a discussion of proposed features and other avenues for future work. Finally, Appendix A gives a complete list of error messages used as output from the parser/classifier, and Appendix B gives the complete code of the scripts used to classify examples and test the parser/classifier.

Chapter 2

Implementation

This project aimed to assist students in learning how to program in Python by making Python syntax errors more detailed and easier to understand. Our goal was to use these simpler, more detailed error messages to improve students' debugging skills, eventually allowing them to interpret Python's original, less detailed error messages on their own.

We decided to focus on syntax errors in Python rather than some other class of errors or some other programming language for a few reasons. Firstly, Python is the language used in most introductory EECS classes at MIT, as well as in most introductory CS courses at top-ranked universities [3]. Secondly, syntax errors in Python are quite common, with 15.3% (in the MIT dataset (Section 1.2.1)) and 28.1% (in Pritchard's dataset [14]) of all errors being syntax errors. Lastly, syntax errors return the most ill-defined, imprecise error messages of the common types we found in the MIT dataset.

There are two basic components to a syntax error message, which we handle separately. Figure 2-1 shows an example `SyntaxError` message. The two basic components are:

1. The error message ("`SyntaxError: invalid syntax`", at the bottom)

2. The location of the error (the file name and line number, and a copy of the faulty line with a caret showing the exact location)

```
> python example.py
Traceback (most recent call last):
  File "ex.py", line 1
    if x = y:
         ^
SyntaxError: invalid syntax
```

Figure 2-1: SyntaxError message output

If the recipient of the error message is using an IDE or other graphical editor (for example, *IDLE*, which is included with the CPython interpreter [2] and is commonly used in MIT classes), this location information is replaced by a highlight on the line where the error occurred. We tackle each of these components separately in creating our system.

In Section 2.1, we describe the script used to classify example faulty code submissions for later use. Section 2.2 describes the parser/classifier we created for use in separating out specific syntax errors. Section 2.3 explains the testing process for the parser/classifier, including scripts we created to manually and automatically check its responses compared to the examples previously classified. In Section 2.4, we explain how this system is integrated into *IDLE*, a graphical editor, for ease of use. Lastly, in Section 2.5, we describe the testing process for the graphical modifications.

2.1 Classifying Example Code

We first examined the MIT database mentioned in Section 1.2.1, comprising code submissions from students in 6.01 and 6.S080, which are two introductory EECS classes. We manually classified 1048 `SyntaxError`-producing student code samples into 12 different categories, including "mismatched parens", "incorrect operator usage", and "missing colon before indented block". The categories are intentionally vague, to maximize ease of classifying submissions.

To make manual classification as quick and painless as possible, we wrote a simple script to recurse through the file tree in which the student code samples were stored, giving the user of the script a descriptive ASCII-art-based interface to use while classifying samples. Figure 2-2 presents screenshots of the user interface. The script presents the student's code in full, with line numbers, along with the full text of the original error produced by Python's runtime. It then prompts the user to enter one of twelve shortcuts for the different error categories, and, if the selected category requires extra information (eg. the "other" category), it prompts the user to provide that extra information (such as the actual cause of the error). The full text of this script can be found in Appendix B.

Based on these classified categories, the three most common errors were those listed above. With description, they are:

"missing colon before indented block": The line before an indented block does not end in a colon. (For example, the line containing the condition in an `if` statement.)

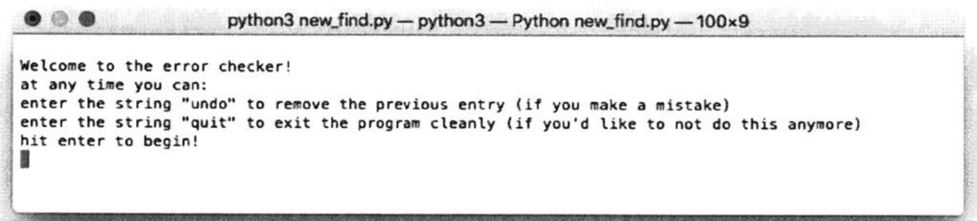
"mismatched parens": Either an opening or closing parenthesis, bracket, or brace is missing.

"incorrect operator usage": An operator is used incorrectly. (For example, using '=' instead of '==' or '&' instead of 'and'. The '=' rather than '==' when used in `if` statements was most common among errors of this type.)

Once we had identified a basic ordering by frequency, we decided to tackle error causes in order from most to least common, while paying attention at the same time to the ease of implementing a checker for each cause.

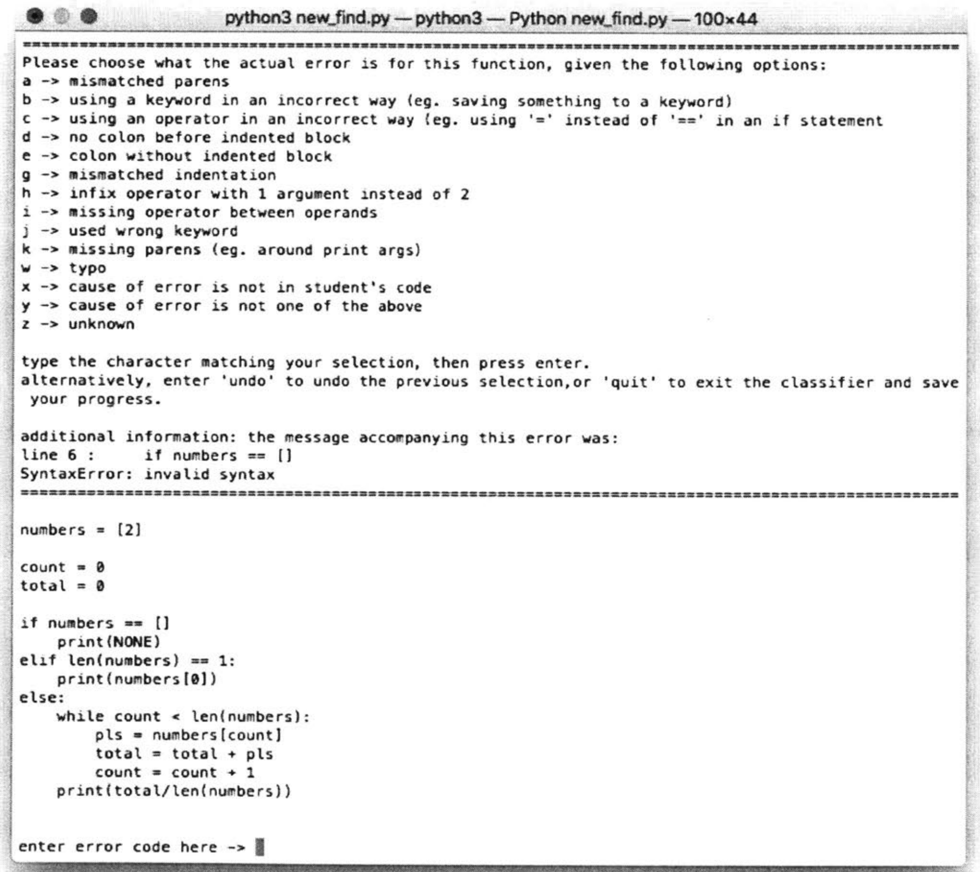
2.2 Parsing and Categorizing Syntax Errors

We started this phase of work by exploring options for a parser to use on our erroring code. We initially looked at how *IDLE* parsed code when it was about to be run, to see if we could simply modify that code to get the desired effect. Unfortunately, the *IDLE* parser would not output the incomplete parse tree (the one that was in



python3 new_find.py — python3 — Python new_find.py — 100×9

```
Welcome to the error checker!
at any time you can:
enter the string "undo" to remove the previous entry (if you make a mistake)
enter the string "quit" to exit the program cleanly (if you'd like to not do this anymore)
hit enter to begin!
```



python3 new_find.py — python3 — Python new_find.py — 100×44

```
================================================================================================
Please choose what the actual error is for this function, given the following options:
a -> mismatched parens
b -> using a keyword in an incorrect way (eg. saving something to a keyword)
c -> using an operator in an incorrect way (eg. using '=' instead of '==' in an if statement
d -> no colon before indented block
e -> colon without indented block
g -> mismatched indentation
h -> infix operator with 1 argument instead of 2
i -> missing operator between operands
j -> used wrong keyword
k -> missing parens (eg. around print args)
w -> typo
x -> cause of error is not in student's code
y -> cause of error is not one of the above
z -> unknown

type the character matching your selection, then press enter.
alternatively, enter 'undo' to undo the previous selection,or 'quit' to exit the classifier and save
 your progress.

additional information: the message accompanying this error was:
line 6 :     if numbers == []
SyntaxError: invalid syntax
================================================================================================

numbers = [2]

count = 0
total = 0

if numbers == []
    print(NONE)
elif len(numbers) == 1:
    print(numbers[0])
else:
    while count < len(numbers):
        pls = numbers[count]
        total = total + pls
        count = count + 1
    print(total/len(numbers))


enter error code here ->
```

Figure 2-2: Screenshots of the classifying script's user interface

the process of being built at the time the error was found), and would instead return merely an error giving us no additional information towards tracking down the source.

We next considered using a pre-built parser or parser generator, one that would return a partially-built parse tree or a parse tree with gaps in the case of a parse error occurring. We looked at three potential pieces of software:

1. *PLY* [1], a fairly complex (but designed for Python) parser and lexer

2. The parser from Kaplan's master's thesis [7], an extremely complex but also extremely customizable parser generator

3. *Parso* [4], a much simpler parser ready-made for the Python grammar, which saves errors as 'error tokens' rather than outputting a partial parse tree from the time of the error

Of these three, we chose *Parso*, as it was the simplest and easiest to set up, while still working for the desired purpose (i.e., showing us the errors in a piece of code in a programmatically explorable way).

The main issue with *IDLE's* parser was that upon finding a syntax error in a section of code, it would return an error and not continue parsing. In contrast, *Parso* inserts an `ErrorNode` at the location of the error (containing the failed-to-parse text), and continues parsing. Thus, it eventually returns a complete parse tree, with any syntax errors replaced by `ErrorNodes`. An `ErrorNode` contains only the code directly causing the syntax error, so that the tree around it (as much as possible) is preserved. Thus, given a 'complete' parse tree from *Parso*, one can iterate over each node in the tree, paying special attention to those denoted as `ErrorNodes`.

Then, given an `ErrorNode`, it is possible to look at the context surrounding and within it to find the root cause of the error. For example, if the actual error is that a coder used '=' instead of '==' in an `if` statement (a common error), then the `ErrorNode` will contain the text '=' and will have as its parent somewhere the clause of the original `if` statement. We can then make a rule saying "if an `ErrorNode` has as a parent an `IfClause`, and the `ErrorNode` contains an operator that is not allowed in an `if` statement (such as '='), then the error is most likely because the coder used a disallowed operator." Similar decision trees can be made for nearly all error nodes.

We also made certain that each newly-constructed error message returned from this parsing/classifying system contained easy-to-understand language. A full list of our output error messages, along with an example syntax error that would trigger each one, can be found in Appendix A.

2.3 Testing the Parser/Classifier

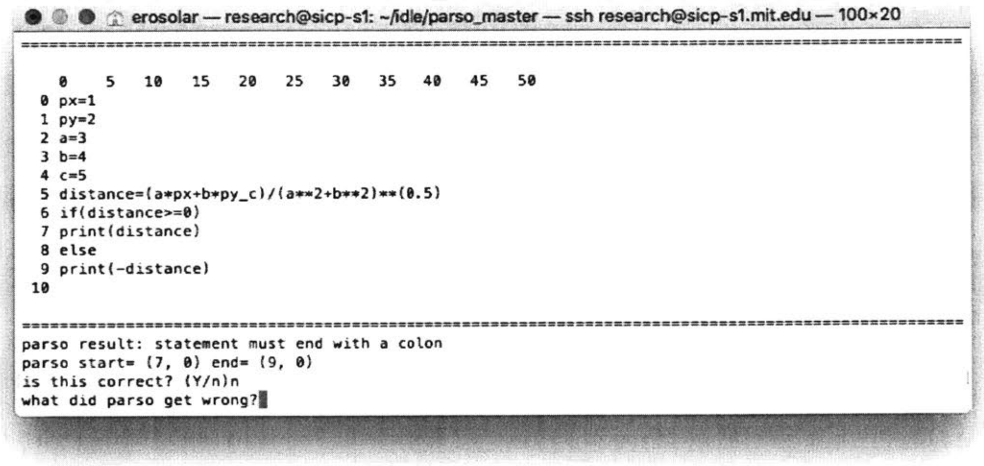
Throughout the process of creating each decision tree, we tested the output from our parsing/classifying system. At first this was done by comparing it to the previously classified code samples, automatically. We created a mapping from error messages to classification, and then simply checked whether a given code sample produced an error message that mapped to its classification. As we added more decision trees and the relative error descriptions became more and more specific, however, it was no longer possible to use the vague classification categories. As such, we modified the classification script so that instead of showing the student's code with Python's response, it showed the student's code with the output from the parser/classifier. Then, we prompted the user to answer whether the parser/classifier's output was correct or not—whether it gave the location and proper description of at least one error that was in the file. If the user said it was not correct, we also requested additional information. Code samples that were marked correct had their identifiers stored in a file so as not to be tested again, and code samples that were marked incorrect had their identifiers stored (along with the extra information) in a separate file so that we could later use that information to fix bugs or add features to the parser/classifier.

In addition to these manually tested cases, the testing script automatically dealt with cases where one of the following things happened:

- An error was returned for code that did not produce an error in Python

- No error was returned for code that produced an error in Python
- The parser/classifier broke while parsing a code sample
- No error was returned for code that did not produce an error in Python

All but the last of these cases were stored as incorrect results, with applicable messages for additional information. (The last was stored in the "correct" file, which did not have any additional information attached.) Figure 2-3 shows an example screenshot of the testing user interface. Similarly to the classification script from Section 2.1, the user interface for this script was optimized for ease of use.


erosolar — research@sicp-s1: ~/idle/parso_master — ssh research@sicp-s1.mit.edu — 100×20

```
====================================================================================================

   0    5   10   15   20   25   30   35   40   45   50
 0 px=1
 1 py=2
 2 a=3
 3 b=4
 4 c=5
 5 distance=(a*px+b*py_c)/(a**2+b**2)**(0.5)
 6 if(distance>=0)
 7 print(distance)
 8 else
 9 print(-distance)
10

====================================================================================================
parso result: statement must end with a colon
parso start= (7, 0) end= (9, 0)
is this correct? (Y/n)n
what did parso get wrong?
```

Figure 2-3: A screenshot of the testing script's user interface

This testing script was re-run every time a significant change was made to the parser/classifier, either adding more decision trees or modifying current ones. A copy of the full code of our testing script can be found in Appendix B.

2.4 Displaying Error Messages

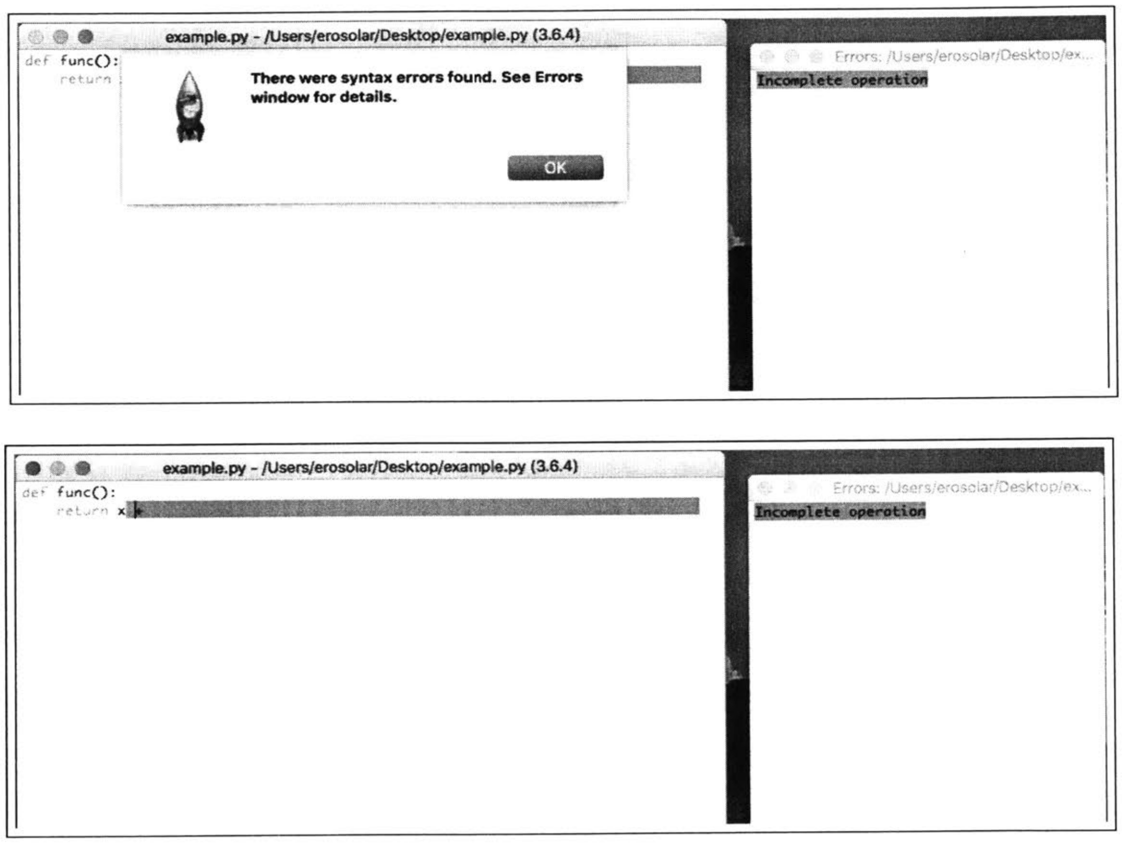
With a mostly complete error description system (our parser/classifier), we started looking at how to integrate it into a graphical interface. Our first task was to decide between *IDLE* and *Detective* as the editor upon which to build our error display.

We decided on *IDLE*, because even though it had a noticeably confusing code base, we felt that novice learners at MIT would be more likely to recognize and know how to use it. (Again, *IDLE* is the default recommendation for students in MIT introductory EECS classes who have not yet chosen their preferred editor. It is also the only syntax-highlighting editor available on the laptops students are encouraged to use in 6.01, one of the more popular introductory EECS classes.) In addition, we had already started investigating *IDLE's* code base, while looking into parser options (see Section 2.2).

Our modified error display system in *IDLE* works as follows. When the user attempts to run a file, it is first checked for syntax errors. In the default version of *IDLE*, if a syntax error is found, the error message returned by Python is displayed as an alert pop-up box to the user, and the location of the error is highlighted. In our modified version, if a syntax error is found, a call is made to the error description system, passing in the erroring file. The error that is returned from the error description system is displayed in a separate window to the right of the editing window, and is highlighted in a color matching the (also highlighted) error location.

A pair of screenshots of this user interface can be found in Figure 2-4. On the left of each screenshot is the editing window, in which we've written a short program with a syntax error. To the right is the error window, displaying our error description system's output. The top screenshot is the state of the GUI immediately after the user attempts to run the file; an alert pop-up box appears, informing the user that syntax errors were found. The bottom screenshot is the state of the GUI after the user dismisses the alert; here we can see the highlighted error, in the same color as the message in the error box. The pop-up box is necessary because, if it were not there, the shell window (not shown in these screenshots) would become focused after an attempt to run the code, regardless of whether that attempt was successful or not. The alert circumvents this, keeping focus on the editor window.

We also implemented support for displaying multiple errors. An example is shown in 2-5. This program has several errors, and two of them have been caught by the error description system. One is highlighted in red, and the other in blue, and each


example.py - /Users/erosolar/Desktop/example.py (3.6.4)

```
def func():
    return
```

There were syntax errors found. See Errors window for details.

Errors: /Users/erosolar/Desktop/ex...

```
Incomplete operation
```

example.py - /Users/erosolar/Desktop/example.py (3.6.4)

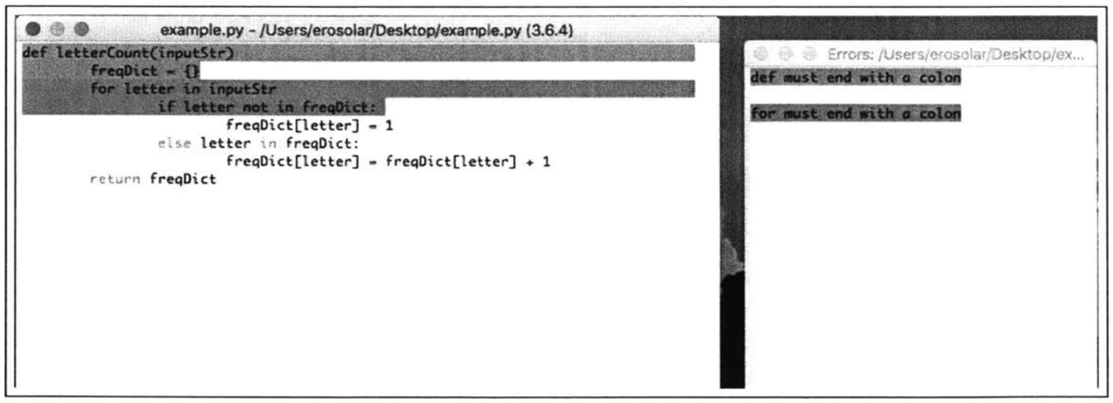
```
def func():
    return x *
```

Errors: /Users/erosolar/Desktop/ex...

```
Incomplete operation
```

Figure 2-4: The updated error display GUI in *IDLE*, before and after dismissing the alert pop-up box.

has a matching highlighted message describing it.



example.py - /Users/erosolar/Desktop/example.py (3.6.4)

```
def letterCount(inputStr)
    freqDict = {}
    for letter in inputStr
        if letter not in freqDict:
            freqDict[letter] = 1
        else letter in freqDict:
            freqDict[letter] = freqDict[letter] + 1
    return freqDict
```

Errors: /Users/erosolar/Desktop/ex...

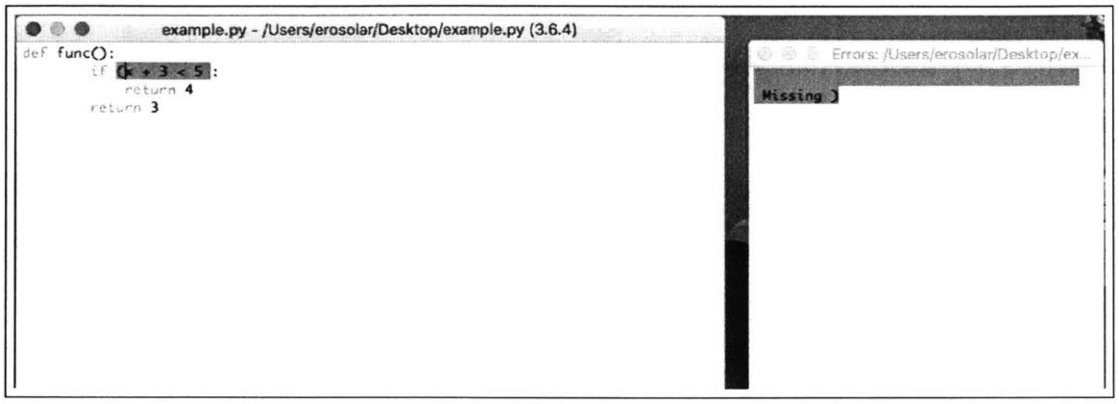
```
def must end with a colon

for must end with a colon
```

Figure 2-5: Multiple error display

The last graphical improvement that we implemented is smart parenthesis finding. Many of the classified errors were due to mismatched parentheses, and we thought that the best way to help novices with mismatched parentheses errors was to have accurate highlighting, to show in what area we guessed the missing parenthesis was likely needed. To implement this, we first modified the error description system to return not only a message but also starting and ending positions. Then, given these positions, we modified the highlighting system in *IDLE* to take both the starting and ending positions, and highlight between them. By default, *IDLE's* highlighting system takes one position, and highlights from that to the end of the word or line. Our errors, and especially missing parenthesis errors, often span more than one word, and so benefit from having highlights with starting and ending positions.

The "missing parenthesis" error is returned by our error description system when a user's parentheses are mismatched. Sometimes, we can look at the surrounding code to compute a compact range where the missing parenthesis should be located. A good example of this is shown in Figure 2-6. Here, the user has forgotten the closing parenthesis in the condition for their `if` statement. Ignoring the missing parenthesis for the moment, the error that our description system would return is that a colon doesn't make sense in the middle of a conditional statement. As we know that the actual error is a mismatched parenthesis, we can ignore the cause of the error and focus on the location. Since the error is that the colon doesn't make sense inside this parenthetical environment, that means the colon is likely supposed to be outside of the parenthetical, and thus that the closing parenthesis is likely between the matching opening parenthesis and the colon. Thus, that area is highlighted in the figure.


example.py - /Users/erosolar/Desktop/example.py (3.6.4)
Errors: /Users/erosolar/Desktop/ex...

```
def func():
    if (x + 3 < 5:
        return 4
    return 3
```

```
Missing )
```

Figure 2-6: A parenthesis error with highlighting.

We implemented similar logic for all missing parenthesis/bracket/brace errors. It does not work in all cases, especially when the error that our parser returns is in a completely different area, or when the error is a legitimate error or otherwise not related to a piece of syntax that doesn't make sense inside a parenthetical context (eg. "incomplete operations" are incomplete operations regardless of whether or not they happen inside a parenthetical context). That said, this parenthesis matching logic does assist in some cases, and is no worse than the default highlighting (which highlights only the parenthesis/bracket/brace that is unmatched), so is a net positive.

Because we had modified the system to place highlights in text in *IDLE*, it was also necessary for us to modify the system to remove highlights from text. The original highlighter module in *IDLE* had a feature where it would remove colorization from a newly edited line, most likely to make it look like a coder had possibly fixed the error causing the highlight. As our new highlighter took a range, and thus could span multiple lines, we added a check to the highlight-clearing function: if a tag that was cleared was one of our error highlight colors, it would also remove that color wherever else in the document it was. That way, when a multi-line highlighted error was edited or fixed, the entire highlight for that error would disappear. Note that this does not break highlighting for multiple errors if displayed, as each error is highlighted in a unique color, to provide a visual mapping between error locations and messages.

2.5 Testing the User Interface

To test the GUI, we manually created test files with common syntax errors and ran them in our development version of *IDLE*, verifying at each step that the GUI acted as we expected and in general was simple and easy to use. We focused on making sure that the error output showed up in the correct location, that the error highlighting was easily visible to the user, and that the GUI reacted to interactions as expected. This testing was much more straightforward than the testing described in Section 2.3, as we were testing improvements made to an existing interface, rather than an entirely new system.

2.6 Summary

In this chapter, we described our approach to classifying sample messages; the construction of our parse-tree-based syntax error describer and the modifications made to `IDLE`, our graphical editor of choice; and the testing of the entire system, piece by piece. The combination of all of these pieces is a graphical editor that, when a syntax error is detected, displays a descriptive error for the specific type of syntax error that has occurred and highlights the portion of code in which the specific error occurred.

Chapter 3

Discussion

In the course of our work on this project, there were a few design choices we made that are discussed in more detail in the first section below. Following that, we present a selection of possible avenues for future work on this and similar topics.

3.1 Alternate Design Choices

We accomplished the majority of our work by making small modifications to different systems (*IDLE* and *Parso*) and combining the results, instead of writing an entirely new system from scratch. We chose to do this for a few reasons. Although writing a new system would allow us to effectively control code quality and make it easier to find and fix bugs in the system, it would require exponentially more work to implement and most of that work would be auxiliary to the main goal of making error messages easier to use. As such, we chose to make modifications to *IDLE* to implement our graphical user interface, and to use *Parso* as a base for our error description service.

Another decision we made was to focus on errors once they occurred—once the error message from Python would have shown up—rather than proactively. While displaying potential errors as the student was in the process of wriitng code might have been more helpful in terms of shortening the debugging cycle, we felt it was important to maintain as small a presence as possible, in order to minimize our impact on the default version of *IDLE*. We wanted not only to help students understand the error messages they received through our system, but also to aid them in getting used to how error messages would occur *outside* of our modified editor. In this way, novices using our editor could more quickly and effectively gain the experience that a more seasoned programmer would use to debug syntax errors given normal error messages. We also wanted to make sure that our system would not become annoying to users, the way a proactive tool might.

In the beginning of this project, we decided to focus on syntax errors in Python. Using the MIT dataset as well as the one used in Pritchard's study [14], we had identified that this type of error is relatively frequent, and by examining syntax error messages as compared to other types of errors in Python, we knew they had relatively uninformative error messages. It was the combination of these two reasons that led to our decision to focus on syntax errors, and not either one on its own. In the MIT dataset, for example, there were other errors ("type", "attribute", and "name", specifically) that occurred more frequently and that we could have focused on, theoretically leading to a bigger impact on our group of users. We chose not to do this for one main reason: we felt there was more room to improve syntax errors than the other types. As we stated in Section 1.1, Python syntax errors specifically are vague, and often do not give any information about the cause of the error beyond a possible location. In comparison to this, type, attribute, and name errors (the three more common types of error listed above) each have more detailed error messages that give the name of the variable that had an issue, a description of the issue, and the location of the specific use of the variable that caused the issue. We felt that the improvements we could make to these errors would mainly be an explanation of what could have caused the issue, with possible highlights on surrounding code that might help explain. In total, however, the work that could be done and the possible improvements to be made were fairly negligible, leading us to choose Python syntax errors as our focus.

3.2 Future Work

In this section, we list a selection of possible avenues for future work in this area.

3.2.1 Runtime Errors

One thing we were planning to do during the course of this thesis project was extending our error description system to improve the error messages of runtime errors as well as syntax errors. We didn't eventually implement this, because catching runtime errors would require more infrastructure than the simple "parser with decision trees" approach we used for syntax errors. However, it would be an interesting course of work—for example, one could highlight the original declaration (or most recent declaration, semantically) of an error-causing variable, hopefully giving a partial explanation as to why the error occurred. For a type-based error, for example, the system could explain (given the declaration) the original type of the variable and the operation's expected vs. actual types, to explain why the type-based error occurred, similarly to *Detective's* [6] descriptions of type- and name-based errors. For an out-of-bounds error, as another example, the system could keep track of the length of an array and explain how it was changing over time, to give the student an idea of why a particular access was out of bounds.

3.2.2 Proactive Checking

Another future course of work could focus on writing a proactive system to detect errors. Our system focused on catching errors reactively, once Python had detected an error. A proactive system would look for errors as the user typed or when the file was saved, giving feedback at times other than when the code was run. This would likely, as discussed earlier, shorten the debugging cycle, allowing for quicker feedback to the user. Of course, care would need to be taken to ensure the system did not become annoying to a user, thus detracting from the joy of programming, but there is likely a lot of data that could be used to assist a programmer in working more effectively—in addition to catching bugs before they actually cause errors. For example, the system could keep track of variables with the same name, so that when one copy of a variable's name was changed, the others could be suggested as needing changes. Alternatively, the system could keep track of the programmer's copy register, pointing out copy-pasted code blocks as being locations where variable names might need to be changed, or recommending those code blocks for abstraction into functions, to discourage copy-pasting in general.

A system relying less on typing information could instead focus on static analysis of the code as it is written, finding variables that aren't declared when accessed, operations that don't have the expected type signature (int + string, for example), or abnormal indentation (as an indicator for possibly wrong control flow), and pointing those situations out to the programmer as possible bugs—similar to linting system.

In general, when considering a proactive checking system, it is important to make sure that the system is helpful rather than annoying. To this end, one must consider how the user interface of such a system would look, and to include ways for a user to ignore specific instances of, or disable completely, functionality that they might consider unhelpful or annoying. A proactive system should be as easy to use and as non-frustrating and non-annoying as possible, lest it be disabled entirely.

3.2.3 Data and Analysis

We made some testing-related modifications to our updated version of *IDLE*, to record data from a few test runs that were made. A similarly modified editor could be used to generate larger datasets for more in-depth examination than that described in Section 1.2.1. A modified version of *IDLE* that could save separate versions of a file when saved in the editor, or when run in the editor, could produce a large body of data (in the form of possibly error-producing code samples) that might give more insight to how likely beginning coders are to encounter errors, what types of errors are most likely, or what fundamental misunderstandings are common among those beginning coders, among many other avenues of analysis.

3.3 Conclusion

In this chapter, we've discussed some choices that we made in the course of implementing our error description system, and we've given some examples of future work that could be undertaken. There are several interesting avenues for future work inspired by this project, in a few different directions.

Overall, through the course of this project, we created a system that improves Python syntax errors by more distinctly categorizing them, giving descriptive messages for each category, and displaying those messages, along with highlights, in an improved graphical editor. We thoroughly tested this system, and are confident that it will improve novice programmers' experience in debugging programs in Python.

Appendix A

Error Message Output

Error Message	Code Example
Invalid Python. Likely copy/pasted header from the terminal.	Python 3.6.4 (default, Mar13 2018, 14:40:33)
++ is not a valid operator	`x++`
-- is not a valid operator	`x--`
{} is for dictionaries only	`if (x < 3) {...`
function definition needs ()	`def foo:`
function definition must have a name	`def (x, y):`
function arguments must be variables	`def foo(3):`
The keyword `pass` should not be followed by anything on the same line	`pass x = 3`
Missing parentheses in call to `'print'`	`print x`

Missing corresponding `try`	`# no 'try:' here`
	`except:`

Can't assign value to variable in `return` statement	`return x = 3`
Can't assign value in `return` statement; attempting comparison with invalid comparator	`return 3 = 3`
Incomplete variable assignment	`x =`

Error Message	Code Example
Variable types do not need to be declared	`int x = 3`
Operator not valid comparison	`if x = 3:`
`if`[1] statement must end with a colon	`if x = 3`
Incomplete operation	`x = 3 +`
two var names in a row	`this thing = 3`
Missing operator; need * for multiplication	`x = 3y`
Missing operator between numbers	`x = 3 3`
Missing operator	`x = "Hi" y`
Missing operator	`x = 3(y + 2)`
Malformed for loop	`for x == 3:`
Malformed `'for'` loop; missing `'in'` in for loop	`for x range(5):`
Malformed `for` loop; uses comparison instead of iteration	`for x < 5:`
Malformed `'for'` loop	`for x in range(1:5)`
Missing corresponding `if/elif`	`# no 'if:' or 'elif:' here`
	`else:`
Should be `'elif'` instead of `'else if'`	`else if x > 3:`
Comparison operations must have at least two operands	`if x > 3 and < 1:`
! is not an operator in Python	`if x:!`
Misplaced or misused colon	`x = 3 + y:`
Malformed string; missing closing "[2]	`x = "a string`
`'elif'` requires a condition	`elif:`
Misplaced or misued colon	`elif: x > 3:`
`'if'` requires a condition	`if:`
`'else'` should not have a condition	`else x > 3:`

[1]This would be replaced with whatever colon-requiring statement the student used
[2]replaced with "opening" and '' as appropriate

Error Message	Code Example
Missing ([3]	`x = 3 + 4)`
unindent does not match any outer indentation level	`if x > 3:`
	`    x = x + 3`
	`  print(x) # error here`
variable cannot be parameter and global	`glob = 3`
	`def foo(x, glob): # error here`

[3]replaced with one of),{,},[,] as appropriate

Appendix B

Scripts

Listing B.1: Hand-Classification Script

```
"""
Used to hand-classify code samples as containing a given syntax error.
Samantha Briasco-Stewart (erosolar@mit.edu)
"""
import os
import re
from enum import Enum
import json

BUILTIN_TYPES = ["'list'", "'dict'", "'int'", "'str'"]
def fix_message(matchobj):
    """
    sanitizes error message by removing specific variable names,
    numbers, function names, etc, and replacing with (constant)
    generic replacements
    """
    fmessage = matchobj.group()
    fmessage = fmessage[:-1]
    def repl(matchobj):
        """
        designed to limit known types to those in BUILTINS,
        so user-made types will be replaced
        """
        if matchobj.group() in BUILTIN_TYPES:
            return matchobj.group()
        return "'x'"

    if fmessage.startswith("Attribute"):
        fmessage = re.sub(r"'[^']*'", repl, fmessage)
    else:
        fmessage = re.sub(r"'[^']*'", "'x'", fmessage)
    # replace named functions with f()
    fmessage = re.sub(r"\w+\(\)", "f()", fmessage)
    # replace things after KeyError with xyz
    fmessage = re.sub(r"KeyError:.*", "KeyError: xyz", fmessage)
    # replace numbers with n
    fmessage = re.sub(r"[0-9]+", "n", fmessage)
    return fmessage


ERR_TYPE_REGEX = re.compile(r"(\w+Error)")
ERR_MESSAGE_REGEX = re.compile(r"(\w+Error[^<]*)<")
def parse_error_things(ftext):
    """
    Looks at a json response from catsoop and parses out the type of
    error, the error message, and a sanitized version of the error
    message (without specific variable names), returning all three.
    """
    # only label syntax errors
    if "SyntaxError" not in ftext:
        return None
    # parse out message included (eg. location) with Python output
    idx = ftext.index("Error")
    matchobj = ERR_TYPE_REGEX.search(ftext, idx-15, idx+20)
    if matchobj is None: # no error found
        return None
    file_error = matchobj.group()
    matchobj = ERR_MESSAGE_REGEX.search(ftext, idx-20, idx+100)
    if matchobj is None:
        return None
    orig_message = matchobj.group()[:-1]
    file_message = fix_message(matchobj)

    return (file_error, file_message, orig_message)


LINE_REGEX = re.compile(r"(line␣\d+[^\<]*)<br\/>([^\<]*)<br\/>")
def parse_error_line(ftext):
    """
    looks at json response from catsoop to parse out the location of
    the error
    """
    # figure out the erroring line of code, and where it is
    ftext_obj = json.loads(ftext)
    loc = None
    resp = ftext_obj["response"]
    if "line" in resp:
        idx = resp.rindex("line")
        matchobj = LINE_REGEX.search(resp, idx-2, idx+150)
        if matchobj is not None:
            loc = matchobj.group(1) + "␣:␣" + matchobj.group(2)
    return loc


class ErrorKeyword(Enum):
    """
    Defines different types of syntax errors
    """
    PARENS = 'a'            # unbalanced parentheses
    KEYWORD = 'b'           # incorrectly used keyword (eg. 'return = 3')
    OPERATOR = 'c'          # incorrectly used operator (eg. 'if x = 4')
    NEEDS_COLON = 'd'       # no colon where there should be one
    BAD_COLON = 'e'         # colon where there shouldn't be one
    INDENTATION = 'g'       # mismatched indentation
    INFIX = 'h'             # infix operator with one argument (eg. '1 +')
    NO_OP = 'i'             # missing operator between operands
    WRONG_KEYWORD = 'j'     # used wrong keyword (eg. 'pass' for 'return')
    MISSING_PARENS = 'k'    # missing parens (eg. around print args)

    TYPO = 'w'              # simple typo (no obvious understanding error)
    NOT_STUDENT = 'x'       # cause of error not in student's code
    OTHER = 'y'             # cause of error is not in above list
    UNKNOWN = 'z'           # hand-checker could not determine cause


def user_interface():
    """
    prints the ascii art interface for the user to see when classifying
    a particular code sample
    """
    print("="*100)
    print("Please␣choose␣what␣the␣actual␣error␣is␣for␣this␣function,␣"
            + "given␣the␣following␣options:")
    print("a␣->␣mismatched␣parens")
    print("b␣->␣using␣a␣keyword␣in␣an␣incorrect␣way␣"
            + "(eg.␣saving␣something␣to␣a␣keyword)")
    print("c␣->␣using␣an␣operator␣in␣an␣incorrect␣way␣"
            + "(eg.␣using␣'='␣instead␣of␣'=='␣in␣an␣if␣statement")
    print("d␣->␣no␣colon␣before␣indented␣block")
    print("e␣->␣colon␣without␣indented␣block")
    print("g␣->␣mismatched␣indentation")
    print("h␣->␣infix␣operator␣with␣1␣argument␣instead␣of␣2")
    print("i␣->␣missing␣operator␣between␣operands")
    print("j␣->␣used␣wrong␣keyword")
    print("k␣->␣missing␣parens␣(eg.␣around␣print␣args)")
    print("w␣->␣typo")
    print("x␣->␣cause␣of␣error␣is␣not␣in␣student's␣code")
    print("y␣->␣cause␣of␣error␣is␣not␣one␣of␣the␣above")
    print("z␣->␣unknown")
    print("")
    print("type the character matching your selection , "
          + "then press enter.")
    print("alternatively , enter 'undo' to undo the previous selection ,"
          + "or 'quit' to exit the classifier and save your progress.")
    print("")


# so checker can undo if they mis-classify the previous example
PREV_FPATH = ""
def label_error(ftext, fpath):
    """
    provides main interface to ask user to classify a given file's error
    """
    global PREV_FPATH, LABELED_THINGS
    # get Python error output
    error_things = parse_error_things(ftext)
    if error_things is None:
        return
    (file_error, file_message, orig_message) = error_things
    # figure out the erroring line of code, and where it is
    loc = parse_error_line(ftext)
    # get erroring code
    code = json.loads(ftext)["code"]
    # now ask user what the error was
    # print UI
    user_interface()
    # add information about error
    print("additional information: "
          + "the message accompanying this error was:")
    if loc is not None:
        print(loc)
    print(orig_message)
    print("="*100)
    print("")
    print(code)
    # ask for error
    error_code = input("enter error code here -> ")
    if error_code == "undo":
        # handle undo by removing previous fpath from dict
        LABELED_THINGS.pop(PREV_FPATH, None)
        print("\033c")
        return
    if error_code == "quit":
        # handle quit by raising 'done error'
        print("\033c")
        raise DoneError
    error_enum = ErrorKeyword(error_code)
    other_info = None
    if error_enum == ErrorKeyword.OTHER:
        # handle other by requesting more info
        other_info = input("please provide more information: ")
    # store in dictinoary
    LABELED_THINGS[fpath] = (file_error, file_message,
                             repr(error_enum), other_info)
    # keep this filepath in case of undo
    PREV_FPATH = fpath
    # clear screen for next example
    print("\033c")


def recurse_with_memo(fname, fns):
    """
    runs each function in fns on each file recursively found
    starting at fname (which is assumed to be a directory)
    """
    subdirs = os.scandir(fname)
    for item in subdirs:
        if item.is_dir():
            recurse_with_memo(item.path, fns)
        elif item.is_file() and item.name.endswith(".json"):
            if item.path not in LABELED_THINGS:
                with open(item.path) as f:
                    ftext = f.read()
                    for fn in fns:
                        fn(ftext, item.path)


# without this, exiting would be an issue since we are working many
# levels deep in a recursive program.
class DoneError(Exception):
    """
    Raised when the user wants to stop classifying errors
    Used so we can save the user's progress (so they won't have to
    start over)
    """
    pass


# so we don't ask about a given code sample twice
def load_past_data(fname, output_dict):
    """
    imports (assumed json) data from a 'save file' into a working
    dictionary used to load state saved from previous runs of
    this checker
    """
    try:
        with open(fname) as f:
            output_dict.update(json.loads(f.read()))
    except FileNotFoundError:
        pass


# stores code samples with classification
# syntax of dictionary:
# key = filepath of erroring file
# value = (error type, error message, chosen error keyword, other info
# other info is None if error keyword is not ErrorKeyword.OTHER
LABELED_THINGS = {}
```

```
OUTPUT_FNAME = "labeled_data.json"
def main():
    load_past_data(OUTPUT_FNAME, LABELED_THINGS)
    print("\033c")
    print("Welcome to the error checker!")
    print("at any time you can:")
    print("enter the string \"undo\" to remove "
          + "the previous entry (if you make a mistake)")
    print("enter the string \"quit\" to exit the program cleanly "
          + "(if you'd like to not do this anymore)")
    print("hit enter to begin!")
    input()
    # clear screen and begin!
    print("\033c")
    try:
        recurse_with_memo(".", [label_error])
    except DoneError:
        pass
    except: # if we make a mistake, should still save all the data!
        pass

    # output our saved results to file
    with open(OUTPUT_FNAME, 'w') as f:
        f.write(json.dumps(LABELED_THINGS))
    # some nice stats to encourage the checker
    print("currently:", len(LABELED_THINGS), "things labeled!")

if __name__ == "__main__":
    main()
```

Listing B.2: Description System Hand-Checking Script

```
"""
Used to hand-verify the output of the error description system
Samantha Briasco-Stewart (erosolar@mit.edu)
"""
import os
import json
# a library that parses html into readable text (without tags)
import html2text
# the file that runs the error description system
import testing_ast

# state file for correctly-described examples
KNOWN_CORRECT_FILE = "parso_known_good_fpaths.json"
# state file for incorrectly-described examples
KNOWN_BAD_FILE = "parso_known_bad_fpaths.json"
# set to True to ignore past files and start over
# should be set when changes are made to the error description system
RETEST = False
# dictionaries to store state while checker is running
KNOWN_CORRECT = {}
KNOWN_BAD = {}
# stats for checker - files checked, correct and incorrect counts
STATS = {
    "correct": 0,
    "incorrect": 0,
    "total_files": 0
    }


def recurse_with_memo(fname, fns):
    """copy of this function from classifier script with stats added"""
    global STATS
    subdirs = os.scandir(fname)
    for item in subdirs:
        if item.is_dir():
                recurse_with_memo(item.path, fns)
        elif item.is_file():
            if item.name.endswith(".json"):
                STATS["total_files"] += 1
                with open(item.path) as f:
                    ftext = f.read()
                    for fn in fns:
                        fn(ftext, item.path)


def automated_check_error(ftext, fpath):
    """
    handles automated checking:
    -> if system returns no error where there is one, that's bad
    -> if system errors, that's bad
    -> if system returns an error where there isn't one, that's bad
    -> if system returns no error and there aren't any, that's good
    -> otherwise, return error described for user checking
    """
    global KNOWN_CORRECT, KNOWN_BAD, STATS
    # skip files we've already checked
    if fpath in KNOWN_CORRECT or fpath in KNOWN_BAD:
        return
    try:
        ftext_obj = json.loads(ftext)
    except: # invalid json is a thing (usually because it's empty)
        return
    code = ftext_obj["code"]
    # save code to temp file to run through description system
    with open("temp.py", "w") as f:
        f.write(code)
        f.close()
    # run code through description system
    try:
        tool_result, pos = testing_ast.run_tests("temp.py", "3.6")
        start, end = pos
    except Exception as e:
        # description system caused an error
        # save that error as an incorrect file
        KNOWN_BAD[fpath] = "ERROR: " + repr(e)
        STATS["incorrect"] += 1
        return
    nice_text = html2text.html2text(ftext_obj["response"])
    if tool_result == "Code Seems Correct" and "Error" not in nice_text:
        # probably we didn't find an error where there was none
        KNOWN_CORRECT[fpath] = "Automated: True"
        STATS["correct"] += 1
        return
    if tool_result == "Code Seems Correct":
        # we didn't find an error where there was one
        e_idx = nice_text.index("Error")
        start = nice_text.rfind("\n", 0, e_idx) + 1
        end = nice_text.find("\n", e_idx)
        KNOWN_BAD[fpath] = "Automated: error= " + nice_text[start:end]
        STATS["incorrect"] += 1
        return
    # else, ask user to figure it out
    user_check_error(ftext, fpath, (code, tool_result, start, end))


def user_check_error(ftext, fpath, automated_info):
    """
    handles user checking: displays error, and prompts user for yes/no
    whether given error is correct. if no, asks for a reason.
    """
    code, tool_result, start, end = automated_info
    # format code to give line numbers
    format_code = '\n'.join(["{0:3d} {1:s}".format(x, y)
                             for x, y in enumerate(code.split("\n"))])
    # display code + message to the user, ask for verification
    print("="*100)
    print("")
    print("␣␣␣␣0␣␣␣␣5␣␣␣10␣␣␣15␣␣␣20␣␣␣25␣␣␣30␣␣␣35␣␣␣40␣␣␣45␣␣␣50")
    print(format_code)
    print("")
    print("="*100)
    print("parso␣result:", tool_result)
    print("parso␣start=", start, "end=", end)
    classification = input("is␣this␣correct?␣(Y/n)")
    if (classification == "" or
            classification == "y" or
            classification == "Y"): # ways to say yes
        KNOWN_CORRECT[fpath] = True
        STATS["correct"] += 1
    else: # assume user said no
        actual_error = input("what␣did␣parso␣get␣wrong?")
        STATS["incorrect"] += 1
        KNOWN_BAD[fpath] = actual_error


def main():
    # hack to make sure file handling works correctly
    os.chdir("../verification/")
    if not RETEST: # load old data
        try:
            with open(KNOWN_CORRECT_FILE) as f:
                fulltext = f.read()
                KNOWN_CORRECT.update(json.loads(fulltext))
                correct = len(KNOWN_CORRECT)
        except FileNotFoundError:
            pass
        try:
            with open(KNOWN_BAD_FILE) as f:
                fulltext = f.read()
                KNOWN_BAD.update(json.loads(fulltext))
                incorrect = len(KNOWN_BAD)
        except FileNotFoundError:
            pass

    # hack to make sure file handling works correctly
    os.chdir("../../6S080_labeled_data/")
    try:
        recurse_with_memo("6s080", [automated_check_error])
    except KeyboardInterrupt as e:
        # so we error out nicely (saving progress)
        pass

    # hack to make sure file handling works correctly
    os.chdir("../idle/verification/")

    # save data to files
    with open(KNOWN_CORRECT_FILE, 'w') as f:
        f.write(json.dumps(KNOWN_CORRECT, sort_keys=True, indent=4))
    with open(KNOWN_BAD_FILE, 'w') as f:
        f.write(json.dumps(KNOWN_BAD, sort_keys=True, indent=4))

    print("total␣files␣examined:␣", STATS["total_files"])
    print("correctly␣identified:␣", STATS["correct"])
    print("incorrectly␣identified:␣", STATS["incorrect"])
```

Bibliography

[1] David Beazley. Ply (python lex-yacc). `http://dabeaz.com/ply`, February 2018.

[2] Python Software Foundation. Idle. `https://docs.python.org/3.6/library/idle.html`, March 2018.

[3] Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, page 47, 2014.

[4] Dave Halter. Parso - a python parser. `https://parso.readthedocs.io/en/latest/`, April 2018.

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

[6] Adam John Hartz. Cat-soop: A tool for automatic collection and assessment of homework exercises. Master's thesis, Massachusetts Institute of Technology, 2012.

[7] Jeremy Daniel Kaplan. An interpreter for a novice-oriented programming language with runtime macros. Master's thesis, Massachusetts Institute of Technology, 2017.

[8] Andrew Ko and Brad Myers. Debugging reinvented. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 301–310. IEEE, 2008.

[9] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *Acm Sigcse Bulletin*, volume 37.3, pages 14–18. ACM, 2005.

[10] Michael J Lee and Andrew J Ko. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*, pages 109–116. ACM, 2011.

[11] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings*

of the 42nd ACM technical symposium on Computer science education, pages 499–504. ACM, 2011.

[12] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin*, volume 40.1, pages 168–172. ACM, 2008.

[13] Roy D Pea. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1):25–36, 1986.

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