The Error Behind The Message: Finding the Cause of Error Messages in Python

Tobias Kohn
University of Cambridge
Cambridge, UK
tk534@cam.ac.uk

ABSTRACT

The interaction between a novice programmer, and the compiler plays a crucial role in the learning process of the novice programmer. Of particular importance is the compiler's feedback on errors in the program code. Accordingly, compiler error messages are an important and active field of research. Yet, a language that has largely been left out of this discussion so far is Python.

We have collected Python programs from high school students taking introductory courses. For each collected erroneous program, we sought to classify the effective error, and assess if the student was able to fix the error. Our study is a precursor to providing improved error messages in Python, and assess their effectiveness. As such, we are eventually interested in finding ways to automatically determine the effective error, so as to base the displayed message on.

From our data, we found that a considerable part of students' errors can be attributed to minor mistakes, which can easily be identified and corrected. However, beyond such minor mistakes, a proper error diagnosis might have to be based on a goal/plan analysis of the entire program. Likewise, proper assessment of whether an error has been fixed frequently requires more context than is provided by the program alone.

ACM Reference Format:
Tobias Kohn. 2019. The Error Behind The Message: Finding the Cause of Error Messages in Python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287381

1 INTRODUCTION

Making mistakes and correcting them is an important aspect of learning. A central piece of this process is good feedback to the learner, which then leads to the correction and effective learning process. In programming, students' mistakes manifest themselves as syntax errors or bugs in the program. While syntax errors prohibit a successful compilation and execution of the program, bugs become visible through the program's output. Unfortunately, the feedback given by compilers on syntax errors is often inadequate and provides little assistance for the student to correct the program.

In order to improve the automated feedback produced by compilers, we need to study how good the displayed message matches the actual, underlying error in the program code, as well as if the message then enables the student to successfully correct the error. That is, good and helpful feedback relates both to the student's error, as well as the correction.

We are undertaking endeavours to develop helpful error messages for Python. To that end, we have written a parser that detects and identifies over one hundred different syntax errors [11]. In order to assess the quality of the error messages, we want to develop a metric that measures how well the messages fit the underlying errors and if the messages enable students to correct their code.

As a first step towards a metric, we have collected and analysed about 4000 Python programs with errors from high school students. Based on the data, we sought to distinguish between trivial errors, such as misspellings or inadvertent slips, and errors that might indicate some misconception or problem. Inadvertent slips are to be expected quite frequently even in programs of professionals.

Measuring the effects of error messages on the students is far from trivial, and there is no single, accepted metric. We thus tried to find patterns in the data, which might lead to a reliable automated assessment.

2 RESEARCH QUESTIONS

We have collected 4091 instances of error messages, each with the student's program that led to the error message, as well as the subsequent program the student tried to run. Analysing the data, we address the following questions:

- **(RQ1)** How often are errors due to minor, superficial mistakes, which should be easily fixed when pointed out?
- **(RQ2)** Do the displayed error messages correspond to the actual underlying errors?
- **(RQ3)** Is there any evidence that the error messages have an effect on the students at all?
- **(RQ4)** Can we derive a metric for measuring automatically if a student has fixed a certain error?

RQ1: What is the number of minor mistakes? The study of compiler error messages in education is to some extend based on the premise that the syntax errors are due to misconceptions of the students. While misconceptions are an important source of errors, some syntax errors are simply due to "typing mistakes" or "inadvertent slips" [19].

We assume that students do not profit from long, explanatory error messages in the case of minor mistakes. Rather, the error messages should be kept as short as possible, and act as brief reminder. Moreover, minor mistakes are not related to misconceptions of students. Hence, the amount of minor mistakes leading to a certain error message being displayed should be taken into account when enhancing error messages.

The classification of an error as a minor mistake depends on the student and the exact situation. For our study, we decided to count as a minor mistake what could be fixed with a simple edit, such as, e. g., inserting a character.

RQ2: Do the error messages report the actual underlying errors? As pointed out by i. a., Dy and Rodrigo [8], or McCall and Kölling [13], there is a difference between the true error in the student's source code and the compiler error message (CEM) displayed. The improvement of error detection is aimed at closing the gap between the underlying error, and the CEM. We are therefore interested in assessing if the data shows limitations to how close true errors and their messages can get.

RQ3: Do error messages have an effect on students? During classroom sessions, we observed that some students would not read the displayed error message, but rather focus on the code at the highlighted position in their source code. This seemed particularly true for longer, explanatory messages. If the hypothesis that a significant part of errors is easy to fix when pointed out holds, then the error messages might be irrelevant for this class of errors. On the other hand, for more complex errors, the error messages might not provide enough information for the students to fix the error at all. It is therefore not obvious that the error messages itself play a non-trivial role.

If the error message does play a role, then the collected data should contain samples, where the students' response correlates with the error message, even when alternative modifications to the source code would make more sense.

RQ4: Can we decide if an error has been fixed? When proposing new enhanced error messages, we need to assess if the messages help the students fix the problem pointed out by the message. One possibility is to check if the student's subsequent compilation attempt succeeds, or if and where the student has modified the source code. For our study, we were interested if a student's editing of the source code and the disappearance of an error are a reliable metric.

3 RELATED WORK

It has been noted early on that compilers often produce inadequate error messages of poor quality [5]. Numerous projects have therefore been put forward to address the problem, and enhance or improve the error messages presented to the novice programmers, e. g., [3, 9, 18], often with limited success [4, 6, 14]. More recent research has started to investigate if the displayed error messages actually correspond to the underlying error [8, 13], or if students read and react to error messages [2, 12, 16].

Most research agrees in that a small set of syntax errors in novices' programs make up for the bulk of all errors encountered. This has been established for Java by, e. g. Jackson et al. [10], Denny et al. [7], and Altadmri and Brown [1], and for Python by Pritchard [17]. It thus makes sense to concentrate research on these most frequent compiler error messages (CEMs) to assess if enhancing or improving CEMs has an effect on the students.

Students have difficulties understanding CEMs and relating them to their code. One approach to address this issue is to *enhance* CEMs with additional information about what kind of error could be the cause of the CEM, or how it might be fixed, thus providing the novice with the knowledge an expert might have.

However, evaluating the effectiveness of enhanced CEMs proves to be difficult, and studies have found mixed results. Denny et al. [6] studied the number of compilations attempts (submissions), and whether the enhanced CEMs led to a reduction in submissions with errors, but found no effect. Becker [3] found that enhanced CEMs do have a positive effect on the number of erroneous submissions, however, a later study [4] could not confirm these results. Nienaltowski et al. [14] also studied the effect of enhanced CEMs, but found that longer messages with additional information did neither improve the number of correct answers, nor the response time of students in their questionnaire. Odekirk and Zachary [15] concluded from their study with C programmers that those who received enhanced feedback needed less help from the teaching assistants. Finally, Marceau et al. [12] note that "there is no single metric for the 'effectiveness' of an error message". They propose a metric based on whether the students make a reasonable edit. Their metric requires the judgement from an experienced instructor.

Since enhancing error messages has resulted in little success so far, Barik et al. [2] and Prather et al. [16] have investigated if students actually read the error messages. Using eye tracking, Barik et al. found that, indeed, students spend considerable time on reading CEMs and conclude that reading and understanding error messages is about as hard as reading program code. Prather et al. also found that students read the error messages, but also note that some students were so unfamiliar with the materials that the CEMs were of little help to them.

The need to add typical causes for error messages already hints at the obvious difference between the actual underlying error, and the displayed error message. Dy and Rodrigo [8] investigated "non-literal" error messages: messages that do not match the actual error. They note that inaccurate CEMs are a source of problems for novice programmers. Furthermore, there are errors in novices' programs, whose nature remains ambiguous even to expert programmers. McCall and Kölling [13] clearly distinguish between the *programmer's misconception*, the concrete *error* in the program, and the *diagnostic message* displayed by the compiler. They point out that the compiler error message often depends on the compiler's internal structure, and that there is no simple correspondence between the underlying error, and the displayed message. Depending on the context, the same misconception or error can lead to different CEMs being displayed. On the other hand, different errors can also lead to the same CEM.

Traver [19] presents an extensive study of the problem of compiler error messages. Among other things, Traver remarks that the most frequent errors do not necessarily represent the most difficult errors to fix. Furthermore, errors are not only due to lack of knowledge, or misconceptions, but also due to inadvertent slips. Traver also states that the quality of compiler error messages matters, as adequate messages can help the programmer to not simply make random edits to address the error. Additionally, helpful error messages can reduce the workload of instructors explaining the same messages over and over to students.

4 METHOD

Collection of Data. We made our Python editor publicly available[1]. The editor/parser came with the option to send error reports and anonymised copies of the programs back to us. This option was not enabled by default, but was on an opt-in basis. Each student could decide at any time whether to take part in our study or not.

Upon start up, the editor would generate a large random sequence as unique identifier, and send all reports together with this identifier. This allowed us to track a student's progress during a session. Typically, a session would last for an hour at school, but if the editor was restarted, it would generate a new identifier.

Swiss high schools with participating students teach their programming courses between grades 9 and 12, with the majority taking place in 10th grade. The courses serve as a first introduction to programming, but typically 5 % to 10 % of students in these classes have some prior programming experience.

The introductory classes followed a common curriculum, which is based on turtle graphics. Students are introduced to functions, simple loops, parameters, variables, conditional execution, and lists.

Size of Data. All error reports were collected during one year. During this period, we have collected a total of 5440 user sessions, each containing several subsequent Python programs along with all errors reported to the student. The number of programs within a session, as well as the number of reported errors varied greatly. We extracted 6981 raw error messages from the recordings, but only 4091 were directly relevant for our study (see below).

In order to get an estimate of the number of participating students, we looked at the number of recorded sessions each week. On average, we recorded 121 weekly sessions, with a median of 120, and a standard deviation of $\sigma = 76.34$. The highest number was 298 sessions during one week. When looking at the distribution of weekly sessions, holiday weeks can easily be recognised: the number of submissions dropped then to an average of 26.6 with a median of 31. Counting only the non-holiday weeks, the overall average is 152 with $\sigma = 60.28$, and a median of 130. Based on these numbers, and given weekly classes at school, we estimate the number of participating students in the order of 100 to perhaps 150 students.

Selection of Data. Not all recorded error messages were directly usable for our study. Accordingly, we had to remove some of the collected submissions from our data set, leaving 4091 of the total of 6981 error reports for further analysis.

Our editor only recorded the main programs, but not additional files or modules. Thus, if the error was located inside an imported module, we were unable to inspect the erroneous code.

[1]The IDE is freely available under http://jython.tobiaskohn.ch

Program 1 We classified the extra space in the += operator as a minor misspelling.

```
x = 0                      x = 0
sum = 0                    sum = 0
while x < 100:             while x < 100:
    x += 1                     x+= 1
    sum+ = x                   sum = x
```

Runtime errors were mostly filtered out. While we retained *name errors* and *type errors*, we discarded other runtime errors such as "file not found" or "division by zero". We only kept errors that directly stemmed from syntactic or semantic issues with the code.

5 RESULTS: THE NATURE OF ERRORS

5.1 Minor Mistakes

Among the 4091 error instances, we classified 1233 instances as minor, superficial mistakes or misspellings, hence about 30 % of all instances. Most of these misspellings produced name error messages, as well as various messages indicating not properly closed tokens and statements (see Table 1). That students were able to directly fix 95 % of these error instances supports the idea of minor mistakes, which pose little problems to the students.

Displayed error message (CEM)	[ALL]	[FIX]
Name Error	570	544
Missing ')'	144	132
Missing Comma	109	102
Missing ':'	94	94
Inconsistent Indentation	64	60
Unterminated String	50	48
Total	1233	1174

Table 1: Of the 1233 instances of minor mistakes, 46 % **led to a name error being displayed. Students where then able to directly fix it in over** 95 %**.**

An error instance classified as minor if the underlying error could be corrected by a simple edit, i. e. swapping two characters, replacing a character by a different one, or inserting or deleting a character. However, the code around the minor mistake needed to be correct otherwise. For instance, a missing colon at the end of a line would only count as a minor mistake if the body beneath was properly indented. The juxtaposition of two statements as in Program 13 was not counted as a minor mistake, even though it can be corrected by a single edit. However, such an instance has a high probability to come from a misconception. When in doubt, we chose to rather not classify a particular instance as minor.

Program 2 The displayed error "NameError: name S is not defined." is caused by a simple mismatch of upper-/lower case. The student "fixed" the CEM by filling in a value for the parameter (which we classified as a "rewrite").

```
def hexagon(s):                def hexagon(s):
    for i in range(6):             for i in range(6):
        forward(S)                     forward(100)
        right(60)                      right(60)

hexagon(100)                   hexagon(100)
```

Nonetheless, Program 1 and Program 2 both show instances where the students modified the program to avoid further error messages, but did not fix the actual underlying error. Given the rather trivial nature of the errors, it is unclear as of why the students seemed unable to fix it.

Program 3 Python finds that the name `s` cannot be resolved (raising a "NameError" at runtime), whereas the actual error is due to incorrect indentation of the **`for`**.

```
def square(s, color):
    pencolor(color)
for i in range(4):
        forward(s)
        right(90)
```

```
def square(s, color):
    pencolor(color)
    for i in range(4):
        forward(s)
        right(90)
```

Program 4 Two examples where the student typed a comma instead of a dot. These errors were reported as "NameError", and "TypeError: wrong number of arguments", respectively.

```
for i in range(100):
    x = randint(0, 10)
    numbers,append(x)
```

```
def circle(r, clr):
        ...
circle(0,5, "red")
```

Program 5 The error message "else without if" might be confusing, because there is, in fact, even more than one **`if`**. The true error is the wrong indentation of the **`else`**.

```
    if key in [LEFT, RIGHT]:
        if key == LEFT:
            heading(-90)
        elif key == RIGHT:
            heading(90)
else:
    forward(10)
```

Program 6 Python reports a "name `p` not found" error in the second line. But the obvious intention of this statement is not to define *n*, but rather *p* as the square root of *n*.

```
n = input("Enter␣a␣number:")
n = p * p
if p % 1 == 0:
    print "This␣is␣a␣square␣number"
```

In addition to the minor mistakes, we found another 35 instances of random artefacts, and remarks. Examples of artefacts include line numbers at the start of each line, or small notes and remarks, which have the character of comments. Most of these notes simply read "Exercise 4.2" or similar.

5.2 Actual and Displayed Errors

Python only has a limited set of error categories. In our context, *SyntaxError*, *IndentationError*, *NameError*, and *TypeError* are relevant. On first sight, this small set suggests that each error can easily be assigned to the correct category. However, as the examples in Program 3, Program 4, and Program 5 show, even in this limited range of error categories, the reported error might not be the actual error.

For instance, of the 1462 name errors in the data set, 47 were due to incorrect indentation or scoping. In another 30 instances, the student used the name of a parameter in a function call instead of providing an argument value. We also found 120 cases where the name error was due to missing string delimiters, as in, e. g., `pencolor(red)`.

In the case of "else without if", 11 of the 13 instances were due to wrong indentation. As Python allows loops to have an else clause, the compiler might often not be able to detect a misplaced else (consider Program 5 with a **`while`** at the top).

Errors as in Program 4 are difficult to identify. Python is dynamically typed, which is to say that functions do not specify the type of their arguments. Moreover, it is often not even possible to clearly determine the correct number of arguments, or even know the function's signature in advance. In fact, the turtle module in Python defines all its functions dynamically, which means that virtually no information is available at compile time. In a case like `pencolor(dark red)`, it is therefore difficult for the parser to determine that there are string delimiters missing. There could, for instance, also be a comma missing between "dark" and "red".

Another case where the error cannot be automatically identified is shown in Program 8. Possibilities include that the last line should be indented, that the body of the if-statement is missing entirely, or even that the if-statement should not be part of the function "onClick".

In seven instances, the students had flipped the assignment, and wrote `3 = x` instead of `x = 3`. The underlying error of putting the target of an assignment on the right instead of on the left becomes much more difficult to recognise in cases like `x = y`. Misconceptions can even lead to examples such as shown in Program 6, where two errors ("assignment to right" and "assignment to expression") are combined, and thereby lead to valid, albeit wrong, syntax.

Program 7 In this instance of "NameError: name `s` is not defined.", the student simply removed the offending variable. The entire structure of this program hints at a basic misconception concerning the scope of variables, or the effect of the return statement.

```
def average(a, b):
    s = (a+b)/2
    return s
average(2,6)
print s
```

```
def average(a, b):
    s = (a+b)/2
    return s
average(2,6)
print
```

6 RESULTS: STUDENTS' REACTIONS

6.1 Statistics

Table 2 provides an overview of the most frequent error messages encountered in the collected data. For each instance in our collection, we looked at the students' reaction to the CEM, and applied a metric similar to the one found in Marceau et al. [12]. In contrast to Marceau et al., we were more concerned with *how* the students fixed the CEM, rather than their reaction in general, leading to different categories. [FIX] means that the student has directly fixed the proximate error in a meaningful way (although other errors might persist). [DEL] means that the student deleted the entire problematic passage (deleting, e. g., an entire function). [RWR] stands for rewrites. As discussed below, these are edits where the student has somehow addressed the CEM by rewriting the code, but not actually fixed the error itself. If a student just deleted individual tokens, but left the overall structure, we counted it as a rewrite rather than a deletion. Finally, there were other kinds of edits not captured by these three categories.

Error Category	Total	[FIX]	[RWR]	[DEL]
Name Error: Cannot Find Name	1462	1252	59	97
Wrong/Inconsistent Indentation	608	485	16	82
Type Error	349	287	17	18
Missing Comma or Operator	249	197	3	22
Missing/Mismatched Brackets	240	223	12	2
Total	4091	3428	176	345

Table 2: The most common error categories with students' reactions.

A case as seen in Program 13 was classified as [FIX], because the resulting program worked as expected, even though putting a comma is not necessarily a "good" edit.

6.2 Rewrites

Falling Back on Familiar Techniques. Using new programming techniques can be a challenge to students. The data contains examples, where students have tried to use a probably new programming construct, but failed because of some (syntactic) detail. In these examples the students then fall back to using familiar techniques instead of fixing the syntax error (Program 10, Program 9).

From the perspective of the compiler, it seems that the students successfully addressed the error and fixed the problem. If we want to evaluate if the error message has been helpful, however, the picture changes. Even though the program behaves correctly, the student was clearly unable to really fix the error at hand.

Fixing the Wrong Thing. Fixing a CEM does not necessarily mean that the underlying problem is solved at all. In some cases, the result compiles (and executes) without error, but does not behave as intended. It is possible to discern two basic student behaviours here. In some instances, the students merely removed the offending part (Program 7). In other instances, the students applied some change to fix the CEM, but thereby introduced logical bugs (Program 8).

Program 8 Instead of fixing the error according to the message "there is a body or indentation missing", the student un-indented the if-statement. The program will run without further problems, because *dotsize* is a global variable. Nevertheless, the if-statement never really takes effect.

```
dotsize = 10                      dotsize = 10
def onClick(x, y):                def onClick(x, y):
    global dotsize                    global dotsize
    dot(dotsize)                      dot(dotsize)
    dotsize += 5                      dotsize += 5
    if dotsize > 30:              if dotsize > 30:
    dotsize = 10                      dotsize = 10
```

Program 9 After the message "there is a body or indentation missing", the student unrolled the loop, altogether.

```
                                  draw_figure(100)
for i in range(4):                draw_figure(100)
draw_figure(100)                  draw_figure(100)
                                  draw_figure(100)
```

Program 10 After the message "to call a function, you must add parentheses", the student directly inlined the function's body instead of adding the necessary parentheses.

```
def square500(red):               def square500(red):
    for i in range(4):                ...
        forward(500)
        right(90)                 forward(20)
                                  for i in range(4):
forward(20)                           forward(500)
square500                             right(90)
```

Program 11 Even though the error message "there is an else without an if" is technically correct, it does not capture the actual problem of the unindented break statement. The student obviously fixed the displayed syntax error, but did not address the true issue here.

```
t = 2                             t = 2
while t < x:                      while t < x:
    r = x % t                         r = x % t
    if r == 0:                        if r == 0:
        print "not␣prime"                 print "not␣prime"
    break                             break
    else                              if r != 0:
        t += 1                            t += 1
```

6.3 The Case of "Missing Comma"

A particular case did not receive the necessary attention during the construction of the parser. When, during the parsing process, an expression is immediately followed by another expression, our parser displays the message that there is a comma missing. We did not, however, anticipate that such a case would occur rather frequently and that putting a comma would be wrong most of the time (see Program 12).

Of the 214 instances that the parser classified as "comma missing", putting a comma would have been correct only in 86 instances (40 %). This means that the students got wrong advice in the remaining 128 instances. In 23 of these instances, the students did put a comma, even though this was not correct, and in 9 additional cases, the students put a comma to separate statements (Program 13). In other words, 32 instances in our data set show that the students read the error message and followed its implied course of action.

6.4 Misconceptions

Program 13 shows an example from our collection, where a student reacted to the "missing comma" message by inserting a comma. Python uses the semicolon to separate statements on the same line. In this case, however, the comma works, too, as it forms a tuple of the values returned by the two function calls.

Later on, several programs contained longer tuples of turtle commands, such as, e. g., `fd(80),rt(120),fd(80),`.... However, once programs include more than simple function calls, the chaining as tuples breaks down, as shown by the two error instances in Program 14. In these instances, the comma has become part of the expression and can no longer be used to separate statements.

The samples where a comma is used to chain statements are rather isolated instances in our data set. This example, however, indicates that error messages can, indeed, help students to learn,

Program 12 Examples of where the parser displayed a "missing comma" error message. Only in the first case is the message correct.

```
[(20, 25) (60, 25)]          pen color("red")
forward 100                  input()"your␣name:"
```

Program 13 After the error message "missing comma", the student added, indeed, a comma, arriving from the statement on the left to the tuple on the right. In Python, the tuple is legal code, and will execute as intended.

```
forward(100)right(100)       forward(100),right(100)
```

Program 14 Two instances, where using the comma as separator between statements does not work as intended.

```
print "...", i+=1            x = 0, y = 0
```

and form conceptual theories about the programming language, and its syntax and semantics. When enhancing error messages, care must be taken so as not to build or foster new misconceptions.

7 DISCUSSION

RQ1: What is the number of minor mistakes? With 30 %, minor mistakes make up a significant part of all the errors collected in our study. Almost half of these minor mistakes manifest themselves as name errors. The errors of a missing closing parenthesis, however, are more serious: there are several cases where standard Python keeps parsing the file, and then reports a completely unrelated error at a later location. Improved error detection can go a long way in helping students in such a case.

RQ2: Do the error messages report the actual underlying errors? Reporting the actual errors in the program code is not only hard in the case of failed compiling attempts. Even syntax errors might just alter a program's logic, and still satisfy the syntactic rules. Moreover, different possible errors can lead to patterns indistinguishable to the compiler, as has also been found in [13].

Not all errors lead necessarily to syntax errors and, moreover, a proper identification of errors requires deeper analysis of a program than is usually performed by a compiler. Hence, the compiler might be the wrong tool to identify and report errors in the first place. Finally, some errors remain ambiguous even to humans, without knowledge of the programmer's goals and plans. This finding is in accordance with [8].

Furthermore, notes and artefacts, such as "`exercise 4`", are for a machine virtually impossible to distinguish from actual code. Even more so as the same program might contain such an artefact, and a function named "exercise4".

On the other hand, even when the compiler error message does not reflect the actual error (as, e. g., in Programs 3 to 5), the message might still give enough information about the problem to be helpful. After all, students were able to fix the errors in over 80 % of the collected instances.

RQ3: Do error messages have an effect on students? Our parser issued incorrect error messages in 128 instances. In 25 % of these cases, the students followed the parser's implicit suggestion. Even though the sample size is very small, this indicates that at least some students rely on the CEM displayed to them. Other possible effects such as reducing the time needed to fix a program, or reducing the number of attempts needed to write a correct program were not considered in this study.

RQ4: Can we decide if an error has been fixed? Apart from the difficulty of reliably diagnosing the error in the underlying program, automatically classifying a student's response proves to be extremely difficult. In various instances, the students addressed the CEM rather than the underlying error. Among the strategies employed by students to cope with the error, we found that some students would just remove any problematic parts, or rewrite them in a simpler way, probably more familiar to them. Conversely, in the case of irrelevant artefacts and notes in the source file, deleting the offending part is, indeed, the right course of action. Checking subsequent compilation attempts for the reported CEMs, or the nature of edits seems therefore an unreliable metric.

Deciding if a particular error has been fixed depends highly on additional circumstances. Additional information such as the goal a program is supposed to achieve might be indispensable.

8 THREATS TO VALIDITY

The manual assessment of program code runs the danger of a strong bias or subjective component. Various examples of students' code illustrate the difficulty of properly assessing the exact type of error, the student's true intention, or even the student's response to the CEM. Furthermore, Python only provides rather crude categories for CEMs, making comparisons difficult.

Probably the largest problem is a proper definition of when an error should be considered a minor, superficial mistake as in RQ1. We decided to be as conservative as possible, only counting instances that clearly satisfy a hard rule and do not show a strong likelihood of an actual misconception.

9 CONCLUSION

Helpful compiler error messages (CEM) should help the programmer correct syntax errors in the program code. Assessing the quality of CEMs involves two independent metrics: the representation of the actual error in the code by the message and the information given to the programmer. In particular, a good metric should measure if the student is able to fix the actual error, instead of just reacting to the CEM.

We have analysed about 4000 error instances of Python programs from high school students and found that a considerable part of the errors are due to minor mistakes, such as misspellings. The nature of other errors or the students' responses, however, cannot always be reliably determined, even by humans, without knowing the goals and plans behind the program. Moreover, the compiler is not even able to detect all syntax errors or correctly classify them.

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

REFERENCES

[1] A. Altadmri and N. C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, 2015. ACM.

[2] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin. Do developers read compiler error messages? In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 575–585, Piscataway, NJ, USA, 2017. IEEE Press.

[3] B. A. Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 126–131, New York, NY, USA, 2016. ACM.

[4] B. A. Becker, K. Goslin, and G. Glanville. The effects of enhanced compiler error messages on a syntax error debugging test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 640–645, New York, NY, USA, 2018. ACM.

[5] P. J. Brown. Error messages: The neglected area of the man/machine interface. *Commun. ACM*, 26(4):246–249, Apr. 1983.

[6] P. Denny, A. Luxton-Reilly, and D. Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education*, ITiCSE '14, pages 273–278, New York, NY, USA, 2014. ACM.

[7] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. ACM.

[8] T. Dy and M. M. Rodrigo. A detector for non-literal java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 118–122, New York, NY, USA, 2010. ACM.

[9] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156, Jan. 2003.

[10] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Proceedings Frontiers in Education 35th Annual Conference*, pages T4C–T4C, Oct 2005.

[11] T. Kohn. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. PhD thesis, ETH Zurich, 2017.

[12] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM.

[13] D. McCall and M. Kölling. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8, Oct 2014.

[14] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 168–172, New York, NY, USA, 2008. ACM.

[15] E. Odekirk-Hash and J. L. Zachary. Automated feedback on programs means students need less help from teachers. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 55–59, New York, NY, USA, 2001. ACM.

[16] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. Cohen. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 74–82, New York, NY, USA, 2017. ACM.

[17] D. Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, pages 1–8, New York, NY, USA, 2015. ACM.

[18] T. Schorsch. Cap: An automated self-assessment tool to check pascal programs for syntax, logic and style errors. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '95, pages 168–172, New York, NY, USA, 1995. ACM.

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