115 Ways Not to Say Hello, World!: Syntax Errors Observed in a Large-Scale Online CS0 Python Course

Bryn Jeffries*  
Grok Academy  
Sydney, New South Wales, Australia  
bryn.jeffries@grokacademy.org

Jung A. Lee  
The University of Sydney  
School of Computer Science  
Sydney, New South Wales, Australia  
jlee6778@uni.sydney.edu.au

Irena Koprinska  
The University of Sydney  
School of Computer Science  
Sydney, New South Wales, Australia  
irena.koprinska@sydney.edu.au

ABSTRACT

Online programming courses can provide detailed automatic feedback for code that fails to meet various test conditions, but novice students often struggle with syntax errors and are unable to write valid testable code. Even for very simple exercises, the range of incorrect code can be surprising to educators with mastery of a programming language. This research paper presents an analysis of the error messages from code run by students in an introductory Python 3 programming course, participated in by 8680 primary and high-school students from 680 institutions. The invalid programs demonstrate a wide diversity of mistakes: even for a one-line “Hello World!” exercise there were 115 unique invalid programs. The most common errors are identified and compared to the topics introduced in the course. The most generic errors in selected exercises are investigated in greater detail to understand the underlying causes. While the majority of students attempting an exercise reach a successful outcome, many students encounter at least one error in their code. Of these, many such errors indicate basic mistakes, such as unquoted string literals, even in exercises late in the course for which some proficiency of earlier concepts is assumed. These observations suggest there is significant scope to provide greater reinforcement of students’ understanding of earlier concepts.

CCS CONCEPTS

• Social and professional topics → K-12 education.

KEYWORDS

CS0, Syntax Errors

1 INTRODUCTION

The need to correctly follow a strict syntax presents a barrier to learning a programming language, and learning to program in general. Unlike natural languages, programming languages are very unforgiving, requiring statements to be expressed exactly and with no tolerance for mistakes. For instance, while some languages such as SQL are case insensitive, most are not; and while many languages are largely insensitive to white space, the very popular Python requires correct use of indentation to nest blocks of code.

A challenge for educators with mastery of the programming language that they teach is that it is hard to anticipate the difficulties a novice student will encounter when being taught a programming language for the first time. The educator then risks making the scope and pace of an introductory programming course poorly matched for the intended audience.

This paper reports on the syntax errors observed in a large-scale online introductory Python programming course for primary and high-school students. It identifies the most common errors, and explores the underlying causes for several specific exercises.

This paper as organised as follows. A brief summary of the literature relating to syntax errors is presented in Section 2. The course and the data extracted for analysis are explained in Sections 3 and 4, respectively. The mistakes in students’ invalid code for the first two exercises of the course are explored in depth in Section 5. A broader look at the incidence of different syntax errors across the whole course is then presented in Section 6. A discussion of the results is provided in Section 7, followed by conclusions in Section 8.

2 BACKGROUND AND RELATED WORK

Learning to program requires learning the syntax of a language as well as the underlying concepts. However, educators who have already mastered the syntax, often underestimate how much attention has to be given to syntax errors from the beginner’s perspective [11]. They often focus strongly on the concepts, leaving students to adapt to the syntax through self-practice [4]. Encountering syntax errors can be demotivating, and impede the learning of concepts, so students can be left behind as the course moves on to new concepts before the students have mastered the previous ones. Qian and Lehman [13] argue that educators must build an understanding of students’ misconceptions in order to successfully teach introductory programming.

Some attempts have been made to avoid the struggle of dealing with syntax errors, by using visual block-based programming languages such as Blockly. However, once students are required to progress to text-based programming, the majority of them get
overwhelmed and discouraged by continuous compilation failures. As a result, many students give up programming during this transition phase as they feel incapable of fixing these errors [12]. As an alternative, Edwards et al. [5] propose a ‘syntax first’ approach, showing that university students who participate in many small syntax exercises demonstrate higher exam performance and lower attrition rates compared to a control group.

Several investigations have been conducted into the syntax errors encountered by students in different contexts, with Java and Python being the languages of attention in most recent work. Denny et al. [3] investigated the most common syntax errors in Java, and specifically how long university students spend resolving the different types of errors and if this is dependent on the level of ability. A small number of error types were found to be especially frequent, and while capable students were able to fix many errors quickly, the two most common types (‘Cannot resolve identifier’ and ‘Type mismatch’) took an average of over 70 seconds to fix for students of all levels of performance. More recently, McCall and Kölling [10] developed an error category hierarchy from Java code written by novice programming students, and scored each category by a severity value based upon their frequency and average resolution time. They observed that the most important errors, when ranked by severity, can be overlooked when ranked by frequency alone.

Brown and Altadmri [1] investigated typical syntax and other errors, also in Java, and the correlation of the error occurrence with the educators’ expectations. Two out of the top 5 mistakes students made were found to be syntax errors. It was also found that educators were not always accurate about the most common types of errors — their ranking had only a moderate agreement with what the actual data showed. Similarly, Jackson et al. [6] found only a weak consensus between educators’ opinions about the frequency of eighteen Java mistakes and the actual numbers encountered by novice university students.

Karvelas et al. [7] observed that novice Java programming students’ use of compiler messages could be affected by the way in which the errors were presented within a development environment. In particular, students tended to make use of compiler error messages when they were made more accessible, and manually compiled their code less often when more error information from previous runs was available.

Smith and Rixner [14] investigated errors in Python programs providing insight into the distribution, duration and evolution of errors. The data was collected from three MOOC courses and included a large number of implementations of eight Python functions, written by novice programmers. Analysing the mistakes, it was found that SyntaxError was the most common, followed by NameError and TypeError. These types of errors were also more likely to re-occur in the next versions of the programs. Other types of errors such as IndexError and KeyError appeared less frequently but took longer to correct which suggests that educators should not only teach students how to avoid them but also how to effectively debug programs. SyntaxError persisted for a subset of students, suggesting that educators should identify these students and provide targeted instruction. However, this was not a traditional high school or university course — the student population was very diverse, with students from 180 countries, of an average age of 39 years, and the effect of the demographic differences was not investigated.

Kohn [8] focused on understanding how well the displayed error message in Python matched the actual underlying error in the program, and if it enables students to correct the error. They analysed about 4000 error instances from Python programs written by 150 high school students. The results show that 30% of the errors were minor, such as name errors, and were easy to identify and fix. The compiler messages do not always reflect the actual error but in general are still helpful, providing enough information about the problem, and students were able to correct most of the errors. It was concluded that beyond the minor mistakes, the nature of the other mistakes cannot be easily determined without knowing the goal of the program — a proper error diagnosis tool requires analysing the goal and plan of the entire program.

More recently, Chiodini et al. [2] have curated a list of of programming language misconceptions in Java, Javascript and Python, based upon several university programming courses run over many years.

The existing research demonstrates that the role of syntax errors deserves considerable attention when choosing where to invest effort in helping students to learn to program. Moreover, educators should not rely upon their own preconceptions of where students will encounter difficulties. While most studies have been of university students, it seems reasonable to expect this to be true of younger students.

3 COURSE OVERVIEW

The NCSS Challenge\footnote{https://grokacademy.org/challenge/} is an online programming course that is run for students at participating schools, mostly within Australia. The language used throughout the course is Python 3. There are several streams available, targeting different levels of prior experience. This analysis looks only at the stream for beginners in Python, from the 2019 Challenge. Students were usually enrolled by their teachers. There were 680 participating schools, with 10,561 students and 467 teachers enrolled. Of the enrolled students, 8680 actively participated by running code at least once within the course. Of those students, 84.5% were in high-school grades (Years 7–12), 15.3% in primary (Kindergarten to Year 6), and the remainder undisclosed. For self-declared gender, 30.0% were female, 52.8% male, and 3.7% were other. 1,976 (18.7%) of students had participated in a previous iteration of the course.

The course ran for 5 weeks. Each week, 2 modules were released, with increasingly advanced content. Each module was designed to allow students to work independently, with 10–15 slides of explanatory content and 4 code-writing exercises. Many of the exercises were designed in pairs, with very similar expected solutions, to allow teachers to guide students through the first exercise in-class before students attempted the second exercise on their own. Since teachers often did not have programming experience, a team of 135 volunteer tutors consisting of experienced IT professionals, teachers and university students provided assistance via an in-platform messaging system.

The course initially focuses on traditional terminal-based programs that make use of text input and output via the standard
The work reported in this article examines the terminal run data from the 2019 edition of the beginners-level course. Each terminal run is associated with a specific exercise in the course, and the user running the code (using just a unique ID, with no associated personal data). The record includes the executed Python 3 program source, the exit code of the program, and the recorded data sent to the standard I/O streams stdin, stdout, and stderr.

There were 1,281,068 terminal runs recorded across all 40 exercises in the course, run by 8680 distinct users (78.7% of the enrolled users). Of these terminal runs, 328,476 (25.6%) resulted in an error reported by the Python 3 interpreter, with an error format "<Error Type>: <message>". There were 3309 distinct error messages.

Figure 1 shows the number of students attempting each exercise, broken down by the outcome reached by each student. In 96.1% of cases a student attempting an exercise achieved a successful outcome. 1.6% of attempts resulted in code being submitted for marking, but failing a test. In only 2.4% did students progress no further than simply running code.

5 DEEP DIVE INTO INITIAL EXERCISES

There is a lot of context around why students encounter the syntax errors in a particular exercise, so the code and associated error messages run for several selected exercises were investigated in detail. Firstly, the first two exercises reveal the diversity of errors made by students at the very start of their journey in learning to code. Both exercises are intended to be simple one-line print function calls, with a very small solution space. This has the additional benefit that the students’ code is easy to process to identify common mistakes. For each exercise, all code was first grouped by the associated error message. For each error, each unique code was then manually inspected to determine the main cause of failure. This manual classification process revealed a set of common mistakes, listed in Table 1.

5.1 Week 1 Part 1 Question 1

The very first marked exercise requires students to edit the program

```
print('Hello, Earth!')
```

to instead print the string "Hello, World!". The problem guides them through the process (rewarding with a green tick at each stage) of

1. Changing the string literal from 'Hello, Earth!' to 'Hello, World!'
2. Running the program
3. Submitting the program for marking

The problem editor prevents the user from modifying the surrounding print(), to focus the user on editing only the string literal. An artefact of the exercise construction was that it was still possible to append text after the closing bracket.

Despite the simplicity of the task and the level of scaffolding, 103 students encountered errors when running code for the exercise, through 115 unique programs. There were 29 instances of IndentationError: expected indent, 63 of SyntaxError: EOL while scanning string literal and 23 of SyntaxError: invalid syntax. As shown in Table 1, these were caused by three principal types of mistakes, often present together in a single program.

In Newlines in strings the string literal was split over more than one line — Python does supports multi-line strings, but they must be enclosed in triple quotes, and students are not introduced to this form of string in the course.

In Unmatched quotes the string literal is not correctly enclosed in consistent quotation marks. Sometimes this was because the terminating quote mark was absent, or didn’t match the opening quote mark (‘ and ” are permitted). In some cases a surplus quotation mark was included on one or both ends, perhaps confusing two single quotes (‘’) with a double quote ("), instead resulting in the intended string literal text to sit outside the quotes. One student struggling with an error because of this last case of mistake attempted to fix the problem by introducing more and more newlines between the last two quote characters, as if to hide the offending extra quote from the interpreter — sadly, this resulted in the
same error. It is worth noting that unmatched quotes could trigger
two different error messages, further complicating the debugging
process for students.

Mismatched brackets usually occurred when a student introduced
an additional opening bracket to the print function. For this and
unmatched quotes, it is possible that students accidentally included
extraneous characters when copying text from other sources (e.g.,
earlier slides in the course), without realising their significance.

### 5.2 Week 1 Part 1 Question 2

The second exercise required students to write a program from
scratch (i.e., with no initial code), to print One fish, two fish.
This led to eight common mistakes, shown in the second section of
Table 1. It can be seen that the resulting error message is often very
obscure compared to the cause. Indeed, the most common error was
the very generic SyntaxError: invalid syntax accounted for
1532 errors, across 1366 students.

As with the unmatched quotes in the first exercise, Incomplete
string and Unquoted string suggest an unawareness of the signif-
icance of quote characters in defining a string literal. Similarly,
Quoted command indicates that purpose of quote characters is am-
biguous to the student. It should be noted that Python 2 allowed
the use of print without parentheses, so this mistake could also
stem from prior experience with Python 2 to Python 3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
<th>Resulting Errors</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline in string</td>
<td><code>print('Hello, world!')</code></td>
<td>SyntaxError: EOL while scanning string literal</td>
<td>48</td>
</tr>
<tr>
<td>unmatched quotes</td>
<td><code>print('Hello, world!')</code></td>
<td>SyntaxError: invalid syntax</td>
<td>62</td>
</tr>
<tr>
<td>Mismatched brackets</td>
<td><code>print('Hello, world!')</code></td>
<td>SyntaxError: invalid syntax</td>
<td>55</td>
</tr>
<tr>
<td>Unmatched quotes</td>
<td><code>print('Hello, world!')</code></td>
<td>SyntaxError: invalid syntax</td>
<td>62</td>
</tr>
<tr>
<td>Mismatched brackets</td>
<td><code>print('Hello, world!')</code></td>
<td>SyntaxError: invalid syntax</td>
<td>55</td>
</tr>
<tr>
<td>Misspelled function name</td>
<td><code>print(&quot;One fish, two fish&quot;)</code></td>
<td>NameError: name '---' is not defined</td>
<td>200</td>
</tr>
<tr>
<td>Incorrect function name</td>
<td><code>write('One fish, two fish')</code></td>
<td>NameError: name '---' is not defined</td>
<td>29</td>
</tr>
<tr>
<td>Missing parentheses</td>
<td><code>print('One fish, two fish')</code></td>
<td>SyntaxError: Missing parentheses in call to 'print'</td>
<td>677</td>
</tr>
<tr>
<td>Accidental whitespace</td>
<td><code>print ('One fish, two fish')</code></td>
<td>IndentationError: unexpected indent</td>
<td>246</td>
</tr>
<tr>
<td>Wrong brackets</td>
<td><code>print[('One fish, two fish')]</code></td>
<td>TypeError: 'builtin_function_or_method' object is not subscriptable</td>
<td>25</td>
</tr>
<tr>
<td>Quoted command</td>
<td><code>print ('One fish, two fish')</code></td>
<td>TypeError: 'str' object is not callable</td>
<td>7</td>
</tr>
<tr>
<td>Incomplete string</td>
<td><code>print('One fish, two fish')</code></td>
<td>SyntaxError: invalid syntax</td>
<td>310</td>
</tr>
<tr>
<td>Unquoted string</td>
<td><code>print(one fish, two fish)</code></td>
<td>SyntaxError: invalid syntax</td>
<td>214</td>
</tr>
</tbody>
</table>

Table 1: Mistakes observed in first (top) and second (bottom) exercises, based on manual classification of program code. Absolute number of students encountering each mistake is also shown. Many programs demonstrated more than one of these mistakes simultaneously.

*This was such a common occurrence in previous iterations of this exercise that the editor now displays a hint 'Are you trying to use print without round brackets? This only works in old Python,' when the error occurs, and the error in the terminal output is annotated with a similar message.*
<table>
<thead>
<tr>
<th>Error pattern</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyntaxError: invalid syntax</td>
<td>40771</td>
</tr>
<tr>
<td>NameError: name *** is not defined</td>
<td>14255</td>
</tr>
<tr>
<td>IndentationError: expected an indented block</td>
<td>11997</td>
</tr>
<tr>
<td>IndentationError: unexpected indent</td>
<td>8651</td>
</tr>
<tr>
<td>SyntaxError: EOL while scanning string literal</td>
<td>7529</td>
</tr>
<tr>
<td>TypeError: unsupported operand type(s) for ***: '---' and '---'</td>
<td>7193</td>
</tr>
<tr>
<td>SyntaxError: unexpected EOF while parsing</td>
<td>5369</td>
</tr>
<tr>
<td>IndentationError: unindent does not match any outer indentation level</td>
<td>3377</td>
</tr>
<tr>
<td>TypeError: *** takes *** positional arguments but *** given</td>
<td>2054</td>
</tr>
<tr>
<td>SyntaxError: Missing parentheses in call to 'print'</td>
<td>1505</td>
</tr>
<tr>
<td>TypeError: Expected positive integer or float</td>
<td>907</td>
</tr>
<tr>
<td>ValueError: invalid literal for int() with base 10: '---'</td>
<td>868</td>
</tr>
<tr>
<td>SyntaxError: can't assign to literal</td>
<td>771</td>
</tr>
<tr>
<td>ValueError: Unknown colour name '***'</td>
<td>715</td>
</tr>
<tr>
<td>TypeError: '---' object is not '---'</td>
<td>653</td>
</tr>
<tr>
<td>TypeError: Expected an RGB triple or string color name</td>
<td>515</td>
</tr>
<tr>
<td>All 984 other (&lt; 500 instances each)</td>
<td>5230</td>
</tr>
</tbody>
</table>

Table 2: Syntax errors encountered by students in the course, aggregating errors with common patterns. One instance is one or more occurrences of that error in a given exercise for a distinct student.

6 WHOLE-COURSE ANALYSIS

To understand the prevalence of different error messages over the whole course, the <Error Type>: <message> signatures from all error messages were inspected manually to identify similar messages, and 42 regular expressions were written and applied to group the messages where a common pattern was evident, reducing the set of distinct messages to 113. Where further investigation was required, we used the stack trace that accompanies each error message, indicating the line and character position at which an error was detected, to identify common causes of errors.

Table 2 shows the list of the most common error signatures, ranked by frequency. To avoid skewing numbers due to a single student repeatedly running the same code and triggering the same error multiple times, each error is only counted once for each student encountering it in a particular problem. 80.8% of all the recorded errors can be accounted by the first six signature patterns. The sixth most common error, "TypeError: unsupported operand type(s)", forming 6.2% of all recorded errors, occurred for 95.4% of students participating in a single exercise (Week 1 Part 2 Question 2). This exercise encouraged students to first run the provided code, to observe the error message caused by a deliberate bug that the student then needed to fix by converting a numerical string into a number. The error was rarely observed in other exercises and so was excluded from further analysis.

Figure 2 shows the proportion of students encountering each of the most common error messages (excluding the TypeError mentioned above) for each exercise in the course. As shown in Figure 1, the number of students attempting each exercise declined steadily throughout the course, with 21.5% of the active users running code in the final exercise. All values are therefore presented relative to the number of students attempting that exercise. Each error type is discussed in turn below.

Indentation Errors. The three forms of IndentationError listed in Table 2 are aggregated in Figure 2 since they share similar distributions and occur for similar reasons. Beginning in Week 2, it can be seen that their frequency becomes significant, and are increasingly frequent in the final weeks. This corresponds to the introduction of constructs such as branching (introduced in Week 2), loops (Week 3) and functions (Week 5) that make use of indentation, exposing students to the quirks of Python’s white-space semantics.

Name Errors. "NameError: name '---' is not defined" occurs frequently for specific problems in earlier modules, and for most of the exercises in Week 5. Exploration of the undefined names in the high-error problems reveals many misspellings of keywords (e.g., ‘imput’). A common trigger was to use an equality operator instead of assignment, e.g.:

```python
book == input('what is your favourite book?')
```
The surge in later exercises reveals issues with scoping (e.g., using a variable before it is declared, or using a function’s local variable in the main code body) which may be important threshold concepts.

**Unterminated Strings.** As observed in Section 5, “SyntaxError: EOL while scanning string literal” occurs when a string literal is not terminated in the line on which it starts. As illustrated in Section 5 this appears to be due to students not having a solid understanding of string literals, an in particular the need to enclose them in quotes. The incidence of such errors declines over Weeks 2 and 3, but spikes again in Week 4. The exercise causing this spike requires students to construct a formatted string literal, using Python’s “f-string” notation⁴. While this notation was introduced in the previous week, many students appear to have forgotten to include a terminating quote character this time around.

**Invalid Syntax.** The generic “SyntaxError: invalid syntax” is by far the most frequent error message, and occurs consistently from Week 2 onward. Inspection of root causes in specific questions revealed a wide variety of mistakes. In Week 2 Part 1 Question 3, in which a choice of two strings is printed depending upon the value of an input string, 47.7% of these errors were clearly due to incorrect expression of the if ... else construct (e.g., missing colons). At least 27.8% of errors were due to string literals not being quoted, and 16.6% were caused by invalid equality tests, such as “=” (assignment) being used instead of “==”. The bulk of errors encountered in the final week’s exercises were related to the use of functions, but still included a substantial mix of more basic mistakes. In Week 5 Part 1 Question 2, 24.3% of errors were due to using `print` without parentheses, and 11.2% were from unquoted string literals.

### 7 DISCUSSION

Many of the mistakes made by students indicated that there was an expectation that the Python grammar would be akin to a natural language: tolerant of mistakes, and accepting of variations that looked similar (e.g., using different braces or synonyms for keywords). Other mistakes point to a level of tidyness that students.may not be acquainted with, especially the need to properly enclose strings in matching quotation characters and close the parentheses for arguments to functions. Many students were evidently still struggling with some of the earlier concepts, such as string literals, as they attempted later exercises that introduced new concepts such as branching, loops and functions. This suggests that more exercises to reinforce early concepts would be helpful. However, it should be noted that most students attempting an exercise appear to eventually overcome the errors. So while there are a lot of errors in the history, they seem to be useful learning experiences rather than hurdles to progress.

Some of the mistakes observed are naturally also made by older students. For instance, the confusion between “=” and “==” concurs with the “AssignCompare” misconception curated by Chiodini et al. [2]. However, it should be noted that the underlying misconception may well be different in this audience, since participants may have a less developed background in mathematics.

Anecdotally, from conversations with some students taking the course, the pedantic nature of the interpreter can be surprising and frustrating. Understanding and accepting this perceived foible is an important lesson for any student. While syntax-free approaches such as programming Blockly may help lower the barrier to entry for young students, there is a good case for providing more exposure to syntax issues, e.g., through syntax exercises [5] to develop a robustness to encountering errors. It may also be helpful to make students more aware of the role of the interpreter, for instance by presenting a stronger personality for the interpreter, as in the work by Lee and Ko [9].

Most of the error messages listed in Table 2 would be difficult for a novice programmer to understand, let alone make use of in the debugging process. One hypothesis is that students instead fall back to code tracing or comparisons to other similar code in order to identify the nature of their mistake. Ideally, the sources of mistakes such as those listed in Table 1 would be inferred, but the process of analysing the errors in this course quickly showed that this becomes difficult even for a relatively small program of a few lines.

### 8 CONCLUSION

Educators must understand the obstacles to their students’ learning in order to focus their teaching appropriately. The dataset analysed here, of syntax errors encountered by school students participating as novices in an introductory Python programming course, provides a wealth of information on the mistakes students make when first learning to program. It complements related work, notably that of Smith and Rixner [14], through a larger size, and of exercises of a more introductory nature that expose the mistakes of a younger audience. As computing becomes a more common subject in many school curricula, such analysis will become increasingly relevant.

The inspection of student code for very simple programs in Section 5 shows a surprisingly wide variety of mistakes made by students who are acclimatising to expressing instructions in a very tightly defined syntax. The higher-level analysis of Section 6 show that many of the mistakes made in these first exercises persist in later exercises, where they may become much harder to isolate, for students and teachers alike.

Given these findings, it seems natural to ask which errors are "sticky", leading a students to repeat them in future, and which are likely to lead to an attrition in participation. Furthermore, it would be interesting to identify whether different groups of students overcome syntax in different ways, and so may benefit from different levels of support. We plan to explore these research questions in future work.

### REFERENCES


⁴https://docs.python.org/3/tutorial/inputoutput.html#tut-f-strings