UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2023

Title: Improving Python Error Messages for Novices

Author: Mandisa Tunzi

Project Abbreviation: ImprovedErrMsgs

Supervisor(s): Dr Zola Mahlaza

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	20
System Development and Implementation	*0*	*20*	15
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	0
Total marks		**80**	**80**

Improving Text-Based Programming Error Messages For South African Students

Mandisa Tunzi
tnzman002@myuct.ac.za
University of Cape Town
Cape Town, South Africa

ABSTRACT

Python is a widely used programming language that is popular amongst novices and in many introductory programming courses. However, one of the significant challenges that novice programmers face is understanding the language's error messages. As a primary source of feedback, error messages can be a barrier to learning if novice programmers struggle to interpret and respond to them effectively. This paper introduces a template-based approach, rooted in natural language generation, for improving the readability and comprehensibility of Python programming error messages. This method involves designing a set of templates that are handcrafted by domain experts following a set of predefined readability guidelines from previous research. These templates are subsequently used to generate improved error messages for a variety of common Python errors. This approach is evaluated by conducting a user study with first year Computer Science students, recognised as novice programmers. In the study, they are tasked with debugging various Python programmes using either standard or enhanced (either improved or translated) error messages. We measure the time that they spend debugging each task, as well as their perceived effectiveness and preference between the standard and improved error messages. The study also investigates the effect of translating the error messages into a different language spoken by the programmers on the debugging performance (measured in time) of programmers. Our results show that the template-based approach guided by the established readability guidelines improved the readability and understandability of Python error messages, as quantified by user perception and preference for the enhanced error messages. However, our results also show that the improvements in error message comprehension did not result in faster debugging times when error messages were enhanced in English. Lastly, our results demonstrate that translating the improved error messages into isiXhosa had a limited overall impact on debugging speed. Nonetheless, these translated messages proved more effective for medium and hard complexity tasks for participants whose first language was isiXhosa, highlighting the potential benefits of localized error messages for programmers whose primary language is not English.

1 INTRODUCTION

Learning how to program can be a challenging task. An important aspect of this learning process is making mistakes and correcting them. Programming error messages facilitate this aspect by providing programmers with information on where and what went wrong in the code. However, these error messages are known to be cryptic and difficult to understand, causing confusion and frustration among programmers [12]. The work done by Prather et al. [2017] and Barik et al. [2017] provides empirical evidence that programmers spend a significant amount of time trying to read and understand error messages and that this impacts their overall task performance [1, 12]. This highlights the importance of error messages for programmers and the need to improve them. Motivated by this, there has been an increase in research to determine ways to improve programming error messages. A significant body of this literature focuses on 'enhancing' error messages to improve their usability and effectiveness for developers. 'Enhancement' here refers to the action(s) of a tool, to improve, the error message presented by another tool, usually by changing or adding to the content of the error message [2]. In order to enhance error messages, studies formulate guidelines based on either their own research or that of others [2].

In their recent work, Becker et al. [2019] conducted a comprehensive review of existing literature focused on enhancing error messages. Their analysis revealed mixed results regarding the effectiveness of these enhancements in making error messages more usable and beneficial for developers. While some studies, using different methods, demonstrated positive outcomes [3, 12, 15], others did not show the same results [5, 10]. As a result of their review, the authors concluded by suggesting the need for further research to address the following gaps they identified:

- Determining a formal definition for readability as well as a way to assess it.
- Gathering empirical evidence in support of one guideline or related pedagogical practice.
- Determining whether the use of identifiers and external documentation could be used to improve the understanding of error messages.

Following this review, Denny et al. [2021] present their efforts in response to the first point above: Readability. Their work establishes 4 readability guidelines that may be used to transform programming error messages [6], namely:

- Using an economy of words
- Using simple vocabulary
- Removing jargon
- Using complete sentences.

In this paper, we apply these guidelines to investigate whether they improve the readability of Python error messages. We utilize a template-based approach for applying the guidelines and enhancing Python error messages. This

approach is based on the idea of using predefined text structures (templates) with placeholders, to generate natural language utterances that present the errors in a way that conforms to the readability guidelines above. These templates are handcrafted by domain expects who have experience in teaching introductory programming. The placeholders in the templates are later filled with data values that correspond to the input meaning representation of the errors, resulting in the generation of improved error messages for a variety of common Python errors that occur frequently in novices' programs.

Our study also has interest in determining the effect of presenting error messages in an alternative language spoken by the novice programmers. In South Africa, where there exist 11 official languages and English is not the primary language for most individuals [8], a gap in literature exists regarding the potential influence of error message representation solely in English on its readability and comprehensibility for non-native English speakers. We extend our efforts to translate the enhanced templates into isiXhosa and evaluate their effectiveness in generating more readable error messages for isiXhosa-speaking programmers.

We present a user study where the following questions guide our analysis:

- RQ1: Will applying a set of readability guidelines to transform text-based Python error messages in to natural language using a template-based approach improve the readability and comprehension of Python error messages?
- RQ2: Will these enhanced error messages allow first year Computer science students debug code faster than with default error messages?
- RQ3: Will first year Computer Science students debug code more efficiently when presented with enhanced error messages in isiXhosa, another language they speak, compared to using the default error messages presented in English.

The rest of the paper is orgainzed as follows: Section 2 presents the background work. Section 3 reviews the related work on error message enhancement. Section 4 describe the Methods and Materials section detailing the design of our template-based approach for improving error messages. Section 5 discusses the design and procedures of the user study. Section 6 and 7 report and discuss the results of a user study respectively. Section 8 concludes the paper, and finally outlines limitations of the study.

2 BACKGROUND

2.1 Template-based NLG for Error Message Improvement

Template-based Natural Language Generation (NLG) is a technique that uses predefined sentence structures, known as templates, to create natural-sounding sentences. This is done by filling these templates with relevant information to ensure coherence, grammatical correctness, and contextual appropriateness. This method is particularly effective because these templates are designed according to the specific domain and purpose of the NLG system [4].

The template-based approach presented in this study offers a method for improving the readability of Python error messages. This approach involves the creation of predefined templates that align with established readability standards. These templates are designed by domain experts who have experience in teaching introductory programming. Functioning as structured frameworks, these templates incorporate designated slots that are subsequently filled with data values from the encountered error instances.

3 RELATED WORK

This section provides a discussion on the work that informs and motivates the current research.

3.1 Error Message Readability

Readability, referring to how easy it is for someone to read and understand the communicated text [7], has been recognised as an important characteristic that good error messages should exhibit [2]. It plays a key role in determining the usefulness of an error message, as an unclear message is likely unhelpful for programmers trying to identify errors [7]. A recent review of existing literature conducted by Becker et al. [2021] synthesizes theoretical and empirical research in this field, revealing a general consensus on the importance of readability in enhancing error messages. However, the literature lacks guidance on how to achieve or evaluate this readability, leading the authors to call for further research to address this gap.

In response to this call, Denny et al. [2021] conducted a series of experiments to identify factors influencing the readability of error messages. The first experiment involved first-year CS1 students evaluating error message readability in popular programming languages. The results indicated that shorter messages without technical language were perceived as more readable. The second experiment analyzed factors such as word count, technical terminology, sentence structure, and vocabulary, showing their impact on readability. The third experiment utilized predefined criteria derived from the prior experiments, demonstrating a strong correlation with initial ratings from the first experiment. Four guidelines emerged from this study to improve error message readability: avoid technical jargon, use concise language, employ simple vocabulary, and minimize word usage. While a separate study applied these guidelines with a machine learning model and observed enhanced readability [11], the current project aims to replicate these findings using a template-based approach, detailed in the following sections. It seeks to investigate whether the observed effectiveness of the established guidelines in improving the readability of error messages holds true irrespective of the natural language they are presented in.

3.2 Translating Error Messages

To our knowledge, prevailing literature focus on enhancing error messages in English, which may not be the primary or

preferred language for many novice programmers around the world. In multilingual contexts, such as South Africa, where there are 11 official languages and English is not the first language for approximately 91.9% of the population [8], there is a gap to determine whether presenting error messages in an alternative language spoken by the novice programmers can improve their readability and comprehensibility. This translation effort is motivated by the positive results, to non-native English speakers reported from localizing programming languages. Guo [2018] revealed that language-localized (translated) programming languages enhance learning for non-native English speakers [9]. In a study, 24 Italian novices using a localized Scratch version outperformed the English version in learning, engagement, and confidence. Raj et al. [2018] found similar results, reporting improved understanding, confidence, and interest among Tamil-speaking students learning programming in their native language [14]. This suggests multilingual programming languages could enhance novice programmers' learning. However, further research is necessary to evaluate the effectiveness of translating error messages into languages other than English, spoken by novice programmers.

4 METHODS AND MATERIALS

This section elaborates on the precise procedures employed in this study. This includes the collection and preparation of a dataset of Python error messages, as well as the methodology used to understand their structure and presentation to users. These steps establish the foundation for the subsequent stages of the research aimed at improving error message readability.

4.1 Selecting Error Messages and Preparing the Dataset

To initiate the research, a dataset of 101 Python error messages was collected from Prichard's work that focuses on the frequency of Python and Java error messages [13]. We selected the top 101 frequently encountered error messages by novices identified by this previous study. Of the initial 101, 9 error messages were excluded due to either their obsolescence in the latest Python version (version 3.10) used in the current research or because we could not find examples to trigger them. Refer to Table 1 below to see the excluded error messages.

This resulted in a refined set of 92 error messages which were then utilized for the study. To access this dataset, refer to Table 9 in Appendix A, where the data is organized by descending frequency of occurrence among novice users, as documented in Prichard's study [13].

Upon further processing of the data, 32 of the remaining errors were then discovered to either have been adapted to resemble existing errors or have been changed in the newer Python version. Refer to Table 10, in Appendix B to see the previous and current versions of these error messages. This process resulted in a dataset of 76 distinct error messages, which were then used to create 62 distinct error message templates. Some of the error messages are represented by the same template, leading to a reduction in the total number of templates derived from the error messages. Refer to Table 11, Appendix C to see error messages and their corresponding templates.

Table 1: Excluded Errors and Corresponding Exclusion Reasons

Errors not Included	Reason
SyntaxError: Invalid Token	Obsolete
SyntaxError: keyword can't be an expression	Obsolete
NameError: Global name ” is not defined	Obsolete
EOFError: EOF when reading a line	Could not find example
MemoryError:	Could not find example
PermissionError: [Errno #] Permission denied: ”	Could not find example
BlockingError: [Errno #] Resource temporarily unavailable	Could not find example
AttributeError: type object ” has no attribute ”	Could not find example
TypeError: func() expected string of length #, but <TYPE> found	Could not find example

4.2 Understanding the Structure of Standard Python Error Messages

In order to enhance the default Python error messages, the initial step involved a comprehensive analysis of their structure and how they are currently presented to users. To do this, we manually categorized the error messages in our dataset by their types and explained the underlying causes using a combination of Prichard's explanations and Python documentation.

For each of the errors, we deliberately created erroneous Python code examples that would trigger the specific error messages. For instance, we intentionally ran the code ***"print (Mandisa)"*** in PyCharm to provoke a NameError along with its associated error message.

This process helped in understanding the structure of Python error messages, including the positioning of important traceback information such as the filename, line number or module in which it occurred. This process subsequently helped in generating templates for the original Python error messages.

These templates were carefully designed to replace error-specific details with designated placeholders while retaining the core structure of the error message. Consider the following sample error message that a Python editor might generate:

```
Traceback (most recent call last):
    File "example.py", line 1, <in module>)
        print (Mandisa)
    NameError: name 'Mandisa' is not defined
```

In response, we developed a corresponding template structured as follows:

```
Traceback (most recent call last):
  File "[FILENAME]", line [LINENUMBER], <in module>)
      [OFFENDING_LINE]
  [ERROR_TYPE]: name '[VARIABLE]' is not defined
```

In this template, information that varies with each error message is replaced with specific placeholders enclosed in square brackets. These placeholders include:

- [**FILENAME**]: Signifying the file name where the error occurred.
- [**LINENUMBER**]: Indicating the specific line number within the file where the error was triggered.
- [**OFFENDING_LINE**]: Representing the line of code that caused the error.

These three components constitute the initial three lines of every Python error message. The final line of the error message is unique, featuring different placeholders depending on the specific error type. For instance, the placeholder "ERROR_TYPE" in the template signifies the category of the error, such as 'SyntaxError' or 'NameError'. The placeholder "VARIABLE" encompasses names of variables, modules, and attributes enclosed in single quotes. Other key placeholders for the error message dataset used in this study include:

- [**TYPE**]: Denoting Python type names like 'int', 'float', 'str' within the error message.
- [**FUNCTION**]: Indicating function calls.
- [**OPERATOR**]: Representing various operator symbols like '+', '-', '*'.
- [**NUMBER**]: Portraying numeric values.
- [**VALUE**]: Denoting numeric values enclosed in quotation marks (string numeric values).
- [**SUGGESTION**]: Conveying sentences, actions, or recommendations.

By utilizing these distinct placeholders, we successfully created a total of 62 unique templates. These templates were subsequently presented to participants for improvement using an established set of readability guidelines.

4.3 Creating the Improved Templates

The process of enhancing the templates involved two phases. During the initial phase, the 62 standard error templates were compiled into an online survey designed to improve the templates by leveraging the insights of former and current first-year programming instructors and tutors. To ensure broad participation, the survey was distributed across multiple platforms, including LinkedIn, whatsApp and on Amathuba, a learning management system (LMS), where UCT first year Computer Science tutors could be reached. In addition to the templates, the survey included example error messages generated by the templates and explanations detailing the specific contexts in which these errors arise in Python programmes. For example, for the NameError template, the following information was provided:

- **MESSAGE EXAMPLE:**

  ```
  File "example.py", line 1, in <module>
  print(Mandisa)
  NameError: name 'Mandisa' is not defined
  ```

- **EXPLANATION:**

  ```
  This error typically occurs when you at-
  tempt to use a variable that has not been
  defined or assigned a value.
  ```

- **TEMPLATE:**

  ```
  File "[FILENAME]", line [LINENUMBER], in
  <module>
  [OFFENDING_LINE]
  [ERROR_TYPE]: name '[VARIABLE]' is not d-
  efined
  ```

Participants were further guided by established readability guidelines that are under investigation. The explanations provided in the survey for adhering to those guidelines were as follows:

- Use simple Vocabulary: Use plain and straightforward language that is easily understandable to a wide range of users.
- Use economy of Words: Make the error message concise by using the fewest words necessary while still conveying the necessary information clearly. No fixed target is placed on the number of words that a message should include, but rather messages should use as many words as are necessary to convey the required information.
- Remove Jargon: Ensure that technical terms or specialized language are avoided, allowing users without advanced knowledge to understand the error message easily. For example, "EOF" is considered jargon because a novice programmer is not expected to know what it means.
- Use complete Sentences: Write the error message as a complete sentence to enhance readability and comprehension. A complete sentence includes a subject, predicate, and independent clause.

The subsequent phase of refining the error message templates involved evaluating the different responses received from different participants. The objective was to identify the templates that best adhered to the four readability guidelines under investigation. We manually conducted this selection process.

The survey collected two responses out of a total pool of at least 56 tutors available on the Amathuba LMS. The number of tutors reachable via WhatsApp and LinkedIn remains unknown. The first respondent deviated from the anticipated approach of modifying the existing standard error message templates. Instead, they briefly outlined suggested improvements for the error templates. For instance, a response included, "Enhance specificity regarding error location." These insights were provided for the initial five templates within the survey, after which the respondent submitted the survey. Conversely, the second respondent complied with the anticipated approach and produced new templates by rearranging and adding to the standard error message templates provided. It was decided that these templates most effectively aligned with the provided readability guidelines. A total of 44 of the 62 templates were modified by the second respondent, all of which followed a consistent structure spanning over 4 lines. The initial two lines maintained the same wording, while the third line offered an explanation regarding potential causes of the error. This

explanation remained consistent with the information provided in the survey, facilitating participants in crafting new templates. The final line provided the error type followed by a brief explanation or suggestion outlining what might have occurred and suggesting potential steps for rectification.

Applying this structure to the earlier mentioned NameError message, the new template reads:

```
There is an error within the code in file: [FILE-
NAME],occurring on line [LINENUMBER].
The suspected line of code causing the error is:
[OFFENDING_LINE].
The problem is that you referenced the name '[VAR
IABLE]' which is not yet defined or assigned a va
lue in your code.
This results in a [ERROR_TYPE]. Python does not
recognise the name and cannot execute the code
before you define it.
```

Following this pattern, we manually transformed the remaining 18 templates. Refer to Appendix D to view the improved templates. The last 18 on the table are the ones improved by the researcher.

The final phase encompassed the translation of these 62 templates into isiXhosa. The templates were translated using Google Translate and were subsequently reviewed by the isiXhosa-speaking researcher to ensure that the translated error messages maintain coherence in natural language, rather than being a mere direct translation. Refer to Appendix D to view the translated templates

4.4 PyPly: The Template-based Python Error Message Generation System

In this section we introduce PyPly, a purpose-built system designed to generate new, improved, and translated error messages. It accomplishes this by extracting error-specific details from standard error messages and seamlessly integrating them into enhanced or translated templates. The system comprises three core components: the lexing mechanism, the parsing mechanism, and the message builder. It takes standard Python error messages as input, extracts error-specific data, retrieves suitable enhanced or translated templates from a database, incorporates the extracted details into appropriate slots within the templates, and finally, generates newly formulated error messages in either English or isiXhosa.

Refer to Figure 1 below for a representation of input and output error messages and Figure 2 for a visual representation of the different components of PyPly and their collaborative operation, detailed below.

4.4.1 The Lexing Mechanism

. This component is responsible for recognising individual elements (parts) of an input error message, such as the name of the file, error type and others.

The process begins with tokenization facilitated by the "ply.lex" module in Python. Here the input message is broken into discrete units known as tokens. Each token corresponds to a specific element within the error message and is tagged with a token type. The lexing mechanism uses a set of predefined regular expressions that match patterns representing

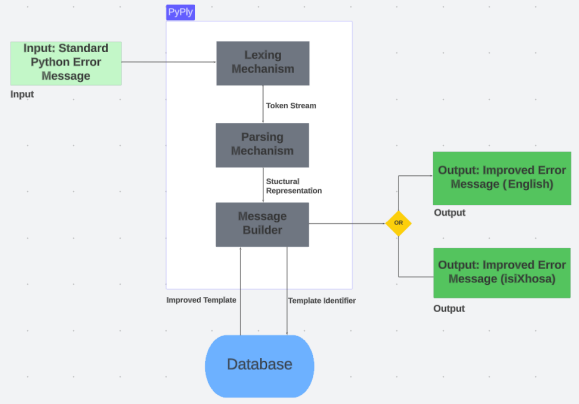

Figure 1: A visual Representation of PyPly: System Components and Flow.

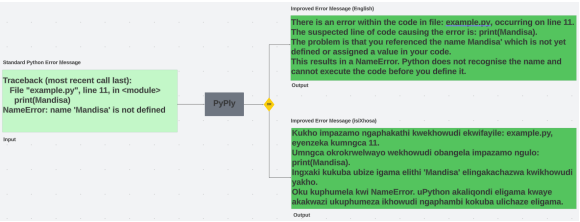

Figure 2: A visual Representation of PyPly: Input and Output Error Messages.

various elements such as types, variables, functions, operators and other information found in a standard Python error message. These regular expressions effectively partition the error message into meaningful segments. For instance, the "FUNCTION" token identifies function calls (strings followed by empty parenthesis), and the "ERROR_TYPE" token categorizes error or warning types.

4.4.2 The Parsing Mechanism

. This component is responsible for the parsing process, comprising essential steps including application of grammar rules and invocation of specialized handler functions.

The parsing process, facilitated by the PLY.yacc module in Python, initiates with the reception of tokens generated during the earlier lexing stage. These tokens are associated with specific token types. The parsing objective is then to comprehend the structure of the error message and extract meaningful components for subsequent processing.

An essential part of the parsing mechanism consists of predefined grammar rules expressed as context-free productions. These rules govern the hierarchal arrangement of elements within an error message. By adhering to these rules, the parser is able to build a structured representation of the error message. The grammar rules encompass the constituents, "message" and "elements" to process inputs with either single or multiple elements in an error message.

In addition to the structural analysis, parsing entails extracting important information from a standard error message and associating it with the corresponding type (element

type). As the parser processes tokens according to the grammar rules, it triggers specialized handler functions specific to each element/ token type. These functions are designed to extract relevant information from the token content and generate tuples, where each tuple pairs the extracted content with its corresponding token type. For instance, "handle_type" function captures Python type names, while the "handle_variable" function captures variable names.

The output of the parsing mechanism is a structured representation of the input error message, This representation takes the form of a list of tuples, each encapsulating the content extracted by the handler functions and their associated token type. Using the NameError example, the structural representation is as follows:

```
('example.py', 'FILENAME'),
(2, 'LINENUMBER'),
('print(Mandisa)', 'OFFENDING_LINE'),
('NameError', 'ERROR_TYPE'),
('Mandisa', 'VARIABLE')
```

The token type corresponds with the placeholders in the template presented previously, signifying where the extracted information should fit into the template.

4.4.3 *The Message Builder*

. This component of the system is responsible for integrating all the pieces together in order to produce the enhanced error messages. It accesses the predefined (enhanced) templates from the database and merges the content extracted in the structured representation into contextually relevant positions. This results in the new enhanced/ translated error message that is then displayed to the user.

5 EVALUATION

5.1 Conducting the User Study

This section describes the participants, design, procedure, measures, and data analysis of our user study, using PyPly, to investigate our research questions.

5.1.1 *Participation*

. The participants of the study were three novice programmers who are currently enrolled and majoring in first year Computer Science at UCT. They all have a few months' experience in programming, having done Python in the first semester from the 13th of February to the 15th of June. The user study took place in August, a month after they started learning Java. They were recruited through convenient sampling, which involves choosing participants based on accessibility and convenience. An invitation to participate in the study was disseminated through a student announcement through Amathuba, accessible only to first year students enrolled in the UCT mainstream Java course.

5.1.2 *Design and Procedures*

. In order to gather essential participant information and establish a baseline for their programming knowledge and skills, each participant was first required to complete a pre-survey. The pre-survey encompassed a programming quiz with ten multiple-choice questions, focusing on introductory Python programming concepts such as selection, iterations, and functions. These concepts were chosen as they aligned with topics covered during their Python learning phase. Additionally, the pre-survey collected background information about the participants, including their first language, proficiency in English and isiXhosa, programming experience, confidence in code writing and debugging, and other pertinent characteristics.

Following the completion of the pre-survey, the three participants were randomly assigned to one of three groups. Participant A was allocated to the control group, which would receive standard error messages. Participant B was assigned to intervention group A (Improved-English), set to receive the English version of the enhanced error messages. Participant C joined intervention group B, (Improved-Xhosa), which would receive the isiXhosa version of the enhanced error messages. They were then asked to engage in a debugging task. This task required them to debug 15 Python programs containing common errors spanning syntax, semantics, and logic. These tasks were designed to be evenly distributed across three levels of difficulty: easy, medium, and hard, as shown in Table 2 below. The tasks were selected and created from a pool of programming exercises for novice programmers available on prominent programming platforms like CodeAcademy and PyNative. The created tasks were modified to introduce a single error per task. The errors were designed such that resolving this error would result in the correct execution of the code, without necessitating changes to any other parts of the code.

Table 2: Tasks, Difficulty Levels, and Associated Errors to Resolve

Task	Difficulty	Error to be Resolved
Palindrome	Easy	AttributeError: lower() function referenced as lowercase()
Factorial	Easy	SyntaxError: Missing colon
FizzBuzz	Easy	IndexError: Single equal sign for assignment instead of double for comparison
ReverseString	Easy	SyntaxError: Used break statement outside of loop
CheckPangram	Easy	SyntaxError: Missing closing bracket
WordFreq	Medium	TypeError: A literal was called like a function
CountSubstrings	Medium	TypeError: Negative sign used in front of a string
AddArray	Medium	NameError: A variable was misspelled
PascalTriangleRow	Medium	TypeError: String used to index an array instead of an integer
MinMax	Medium	UnboundLocalError: Variable referenced before being defined
FibonacciSeq	Hard	IndexError: An out-of-bounds reference to an array
CalcMedian	Hard	ZeroDivisionError: A number was divided by zero
SumOfPrimeFactors	Hard	IndentationError: Inconsistent indentation
Missing Num	Hard	TypeError: Iteration over integer instead of an array
RecursiveFact	Hard	TypeError: Extra argument provided to function

The experiment's procedure was as follows:

(1) The initial 35 minutes were dedicated to participant orientation, wherein they were welcomed, briefed about the experiment, guided through the process of comprehending and signing the informed consent, and the experimental conditions were set. Participants were also given the opportunity to complete the pre-survey and quiz during this phase.
(2) Participants were then allocated a maximum of 80 minutes to work through the 15 debugging tasks. They were provided with explicit instructions to approach each task using a text editor. The sole source of guidance for identifying errors was the error messages presented by the Automarker. Participants were required to submit their solutions to the Automarker, a tool that automatically marks programming assignments by comparing the output of the code to the expected output for evaluation. It verified if the code submitted by participants executed correctly in line with the expected input and

output. Successful execution of all trials for a particular question (task) indicated successful debugging. The Automarker was also tasked with displaying error messages to participants based on their assigned groups. PyPly was integrated into the Automarker such that it could generate the improved and translated error messages for the appropriate groups. Participants completed the tasks in the order they preferred, submitting their attempts whenever they believed they had resolved the bug in a task. The dependent variable of the study was debugging time. It was measured as the time difference between a program's first successful execution (passing all trials on the Automarker) and the first successful execution of the previously completed task (considered as the start time of the current task).

(3) The independent variable of the study was the type of error messages that the participants were exposed to when completing the tasks. The study aimed to test whether the type of error message (standard, improved-English, or improved-Xhosa) had an effect on the debugging performance and feedback of participants.

(4) Out of the 62 templates, 15 were subjected to testing in this user study, encompassing both English and isiXhosa versions. For access to the specific 15 templates that underwent testing, please consult Appendix D. Each of the 15 tasks given to participants was linked to one of the error message templates being evaluated.

(5) After completing the debugging tasks, or after the 80 minutes, the participants were asked to complete a short post-survey that asked them about their perceived difficulty level of the tasks, whether they used any external tools to debug other than the error messages displayed on the Automarker, their preference between standard and improved English/isiXhosa error messages, and their perceived effectiveness of the error messages they used.

The results of this experiment are presented in the next section.

6 RESULTS

This section presents the results of the user study, providing a comprehensive overview of the data collected. We delve into participants' backgrounds, their performance on debugging tasks, and the impact of enhanced error messages in English and isiXhosa on their coding experiences

6.1 Participants' Background Information

Table 3 below summarizes the participants' background based on their responses to the pre-survey. The participants had varying levels of programming experience and confidence in writing and debugging code, with all scores falling in the range of 3 (average) to 4 (proficient) on a five-point Likert scale. All participants reported that standard error messages were often confusing and challenging to interpret. Two of the participants' first languages were English, while one participant's first language was isiXhosa. However, all participants were proficient in both languages. The participants' scores on the programming quiz were 40%, 50%, and 80% for participants A, B, and C, respectively.

Table 3: Participant Background and Coding Experience Based on Pre-survey Responses, Presented in Likert Scale Values

Question/ Characteristic (Pre Survey)	Student A (Control)	Student B (Improved-English)	Student C (Improved-Xhosa)
Computer Science Major	Yes	Yes	Yes
Programming Experience	3	4	4
Confidence in Writing Code	3	4	3
Confidence in Debugging Code	3	3	2
First Language	English	English	isiXhosa
English Profiency	4	5	5
isiXhosa Proficiency	4	5	5
Expecting Benefit from Xhosa Error Messages	No	No	No
% Mark for programming Quiz	40%	50%	80%
Preferred Resources	Course Material and Textbooks	Online forums (e.g., Stack Overflow); Teaching assistants or instructors	Course Material and Textbooks; Peers
Experience with Programming Error Messages	Often Confusing and Challenging to interpret	Often Confusing and Challenging to interpret	Often Confusing and Challenging to interpret

6.2 Debugging Performance

Table 4 below offers an overview of debugging times, quantified in seconds, for each task. These times are measured as the difference between the time of the first successful execution of the current program and the time of the first successful execution of the previously completed task. In Table 5, you'll further find descriptive statistics for the debugging times within each experiment category. This table presents key metrics such as the mean, standard deviation, minimum, maximum, median, and range for each variable and group.

Table 4: Debugging Times (in seconds) of Participant Groups Across 15 Tasks, Organized by Three Difficulty Levels; Zero Indicates Incomplete Tasks

Task	Level	Student A (Control)	Student A (Improved-English)	Student B (Improved-Xhosa)
Palindrome	Easy	92	0	387
Factorial	Easy	53	130	688
FizzBuzz	Easy	105	104	245
ReverseString	Easy	99	167	407
CheckPangram	Easy	81	264	432
WordFreq	Medium	517	0	0
CountSubstrings	Medium	83	237	244
AddArray	Medium	468	700	255
PascalTriangleRow	Medium	132	276	184
MinMax	Medium	774	0	0
FibonacciSeq	Hard	595	609	244
CalcMedian	Hard	534	1369	286
SumOfPrimeFactors	Hard	56	168	0
Missing Num	Hard	1001	1207	0
RecursiveFact	Hard	67	0	519

Table 5: Descriptive Statistics (Mean, Standard Deviation, Minimum, Maximum, Median, Range) for Each Variable by Participant Group

Group	Mean	Std. Dev.	Min	Max	Median	Range
Control (Standard)	310.467	310.682	53	1001	105	948
Improved-English	475.454	446.115	104	1369	264	1265
Improved-Xhosa	353.727	151.052	184	688	286	504

We calculated the percentage change in the average debugging time compared to the control group (Participant A) for Participants B and C:

- Percentage change for Participant B (Improved-English): ((475.55 - 310.47) / 310.47) * 100 ≈ 53.01%
- Percentage change for Participant C (Improved-Xhosa): ((353.73 - 310.47) / 310.47) * 100 ≈ 13.93%

The results indicate that Participant B (Improved-English) took 53.01% more time on average, while Participant C (Improved-Xhosa) took only 13.93% more time compared to the control group. These percentages demonstrate the relative differences in debugging times among the groups.

6.3 Feedback on Error Messages

Table 6 below shows the participants' feedback on the error messages based on their responses to the post-survey. The results showed that the participants perceived the difficulty level of the tasks at an average of approximately 3 on a Likert scale ranging from 1 (very easy) to 5 (very hard). Only participant A (control group) admitted to having used an IDE to debug the tasks, while the others relied solely on the error messages displayed by the Automarker. Participants B and C belonging to the improved-English and improved-Xhosa groups respectively expressed their preference for the improved error messages over the standard ones they have used in the past. The participants also rated the effectiveness of the error messages they used, with participant B giving the highest rating "very effective", participant C giving a moderate rating "Somewhat effective" and participant A the lowest rating "Not very effective".

Table 6: Participant Feedback on Readability and Efficacy of Enhanced Error Messages Based on Post-Survey Responses

Question/ Characteristic (Post Survey)	Student A (Control)	Student B (Improved-English)	Student C (Improved-Xhosa)
Task Difficulty Level	4	3	3
Type of Error Message Received	Standard	Improved - Englished	Improve - Xhosa
External Tool Used to debug Task other than Presented Error Messages	Yes(IDE)	No	No
Standard Error Messages Vs. Improved - English	n\a	Enhanced error messages were equally readable and clear	n\a
Standard Error Messages Vs. Improved - Xhosa	n\a	n\a	Enhanced error messages were more readable and clear
Perceived Effectivity of Error Messages used.	Not Very effective	Very Effective	Somewhat Effective

We discuss these results in the following section.

7 DISCUSSION

Within this section, we examine our study's findings in the context of each research question, delving into the implications and connections between the collected data and the research objectives.

7.1 Research Question 1 (RQ1): Enhancing Readability and Comprehension of Error Messages

Addressing RQ1, which is stated as follows:

- **RQ1:** Will applying a set of readability guidelines to transform text-based Python error messages into natural language using a template-based approach improve the readability and comprehension of the Python error messages?

We analyse the data collected from the participants' responses, particularly focusing on their feedback regarding the readability and effectiveness of error messages.

Our effort to apply a set of readability guidelines to transform text-based Python error messages into natural language using a template-based approach, guided by established readability principles, appears to have had a positive influence on the debugging experience of first-year programming students. Participants exposed to these enhanced error messages (for both the English and translated versions) reported higher levels of readability, clarity, and effectiveness compared to the participant who encountered the standard error messages. Furthermore, they expressed a consistent preference for the enhanced error messages over the standard ones. This preference for the transformed error messages underlines their perceived effectiveness in conveying information and assisting with debugging tasks. These results collectively suggest that the transformation process made error messages more readable and understandable for novice students. However, it's important to note an inconsistency in Student B's responses. While this student initially reported finding the standard error messages difficult to read, they later mentioned that these standard messages were on par with the improved error messages, although they still found the improved versions more efficient. This inconsistency in Student B's responses introduces a level of uncertainty into our initial findings and highlights the complexity of individual perceptions and preferences regarding error message readability and efficiency. Given the limited number of participants who reported on efficiency and the absence of additional data to explain the inconsistencies in Student B's responses, it is important to approach our conclusions with caution and acknowledge the potential variability in user experiences that may impact the overall reliability of our findings.

7.2 Research Question 2 (RQ2): Impact on Debugging Speed

Addressing RQ2, which is stated as follows:

- **RQ2**: Will enhancing error messages in English enable first-year Computer Science students to debug code more efficiently compared to default error messages?

We primarily employ quantitative data related to debugging times and compare the performance of two groups of students: Student A (control group with default error messages) and Student B (improved-English group with enhanced error messages). We also incorporate qualitative data from these participants to gain insights into factors that may influence the observed results.

Comparing the debugging performance of Student A and Student B, we find that Student B, who received improved error messages in English, took, on average, 53.01% more

time to complete debugging tasks. This observation suggests that the enhanced error messages did not lead to faster debugging times; in fact, there was a noticeable increase in the time required for debugging. These results emphasize that while improving error message readability in English may have positively affected comprehension, it did not necessarily translate into faster debugging.

When exploring other factors that may have influenced debugging times, we found that neither the complexity of the tasks nor the participants' programming proficiency had a substantial effect.

Consider Table 7 below showing the average debugging times across the difficulty levels for the two students:

Table 7: Average Debugging Times (in seconds) for Student A (Control) and Student B (Improved-English) Across Various Difficulty Levels

Difficulty Level	Student A (Control)	Student B (Improved-English)
Easy	86.000	166.250
Medium	394.800	404.333
Hard	450.600	838.250

Examining the table, we can make the following key observations:

- In both the control and improved-English groups, we notice a consistent trend where the time required to debug increases as the difficulty level of the tasks increases. This trend suggests that the complexity of the task plays a crucial role in determining debugging time, irrespective of the quality of the error message received. In essence, the specific error message a student receives appears to be inconsequential in this regard.
- Notably, when comparing Student A (control) and Student B (improved-English), we observe that Student A consistently outperforms Student B in terms of faster debugging times across all difficulty levels. Interestingly, participant A also completed all tasks, even though they exhibited lower confidence in their programming skills and achieved the lowest score on the programming quiz, while student B completed 11 of the 15 tasks, all at longer times relative to student A. Therefore, there is no evidence that programming proficiency significantly affected the debugging differences across the students.

However, other factors may have contributed to the difference in debugging times. It is important to note that the debugging time was measured as the time difference between a program's first successful execution and the first successful execution of the previously completed task. This measurement did not account for other factors, such as idle time when students may have been distracted or the time spent familiarizing themselves with the new error messages. Additionally, the time spent packaging tasks and submitting them to the Automarker was not factored into the debugging time calculation.

Student A mentioned using an Integrated Development Environment (IDE), which, while seemingly insignificant with respect to the error messages encountered – as these messages were allocated to the control group and remained consistent – could have still impacted the overall debugging duration. By utilizing an IDE, Student A received instant error message feedback without needing to submit their code to the Automarker, unlike the other participants. This expedited their debugging process, potentially resulting in shorter completion times. Furthermore, Student A's familiarity with the error messages could have played a role in their quicker debugging, in contrast to Student B, who encountered new error messages and may have spent initial tasks acclimating themselves to the format and content of these newly introduced error messages.

7.3 Research Question 3 (RQ3): Language and Debugging Efficiency

Addressing RQ3, which is stated as follows:

- **RQ3**: Will first year Computer Science students debug code more efficiently when presented with enhanced error messages in isiXhosa, another language they speak, compared to using the default error messages presented in English.

Our primary approach involves utilizing quantitative data related to debugging times. We compare the debugging performance of Student A (control group with default error messages) and Student C (improved-Xhosa group with enhanced error messages). Additionally, we incorporate qualitative data from these participants to explore factors that may impact the observed results.

When comparing the debugging performances of Student A and Student C, it is notable that Student C, on average, took 13.93% more time to complete debugging tasks compared to Student A. Although this percentage increase was lower than what was observed for Student B, it does indicate that enhancing error messages in isiXhosa did not significantly improve debugging speed compared to the control group.

However, it is crucial to recognize that the overall averages may not tell the full story. Table 8 below illustrates the average debugging times across different difficulty levels for both Student A and Student C.

As we look deeper into the different difficulty levels, a more nuanced picture emerges. For easy tasks, it is evident that Student C required significantly more time (431.8 seconds) compared to Student A (86 seconds), representing a substantial 402.09% difference. This suggests that enhancing error messages in isiXhosa did not lead to faster debugging for easier tasks. However, the pattern shifts when we examine medium difficulty tasks. Student C's debugging time (227.667 seconds) was notably lower than Student A's (394 .8 seconds), indicating a 73.57% difference. This trend continues into hard tasks, with Student C consistently outperforms Student A, with a debugging time of 349.677 seconds compared to 450.6 seconds, representing a 28.69% differnce. This indicates that enhanced error messages in isiXhosa may have been more effective in aiding debugging for tasks of medium and hard complexity. The initial slower debugging times for Student C, in easy tasks, may be attributed

to factors such as unfamiliarity with the new error message structure. Subsequently, as the student adjusted, their performance improved, resulting in the lowest debugging times.

Table 8: Average Debugging Times (in seconds) for Student A (Control) and Student C (Improved-Xhosa) Across Various Difficulty Levels

Difficulty Level	Student A (Control)	Student C (Improved-Xhosa)
Easy	86.000	431.800
Medium	394.800	227.667
Hard	450.600	349.677

Furthermore, when evaluating qualitative feedback and participant preferences, the evidence strongly suggests that error messages presented in a programmer's first language significantly enhances their readability and comprehensibility to novice programmers. Participants, like Student B (whose first language is English and received improved error messages in English) and Student C (whose first language is isiXhosa and received improved messages in isiXhosa) consistently reported increased effectiveness and clarity. This aligns with their experiences, as they found these messages more effective than standard English-only messages. This linguistic alignment underscores the potential benefits of tailoring error messages to learners' primary or preferred languages, even when a quantitative reduction in overall debugging time average was not observed.

8 CONCLUSIONS

We have explored how enhancing error messages by applying a set of readability guidelines to transform them into natural language or native language would affect the debugging performance and experience of first-year computer science students at UCT. We have conducted a user study with three participants who received different types of error messages for each debugging task they performed. We have measured their debugging time, perceived difficulty, readability, clarity, effectiveness, and satisfaction with the error messages. Our main findings are as follows:

- Enhancing error messages through the application of readability guidelines to transform them into natural language significantly improved their readability and comprehension. This improvement was consistent for both English and isiXhosa error messages, as indicated by participants' feedback on clarity, effectiveness, and satisfaction. Consequently, this positive transformation positively influenced the debugging experience of novice programming students.
- Enhancing error messages in English did not result in faster debugging times for first-year Computer Science students, as the participant who received improved error messages took significantly more time to complete debugging tasks compared to the control group.
- Enhancing error messages in isiXhosa did not significantly improve overall debugging speed compared to the control group. However, a nuanced analysis showed that they were more effective for medium and hard complexity tasks, as the participant in this group achieved faster debugging times compared to their counterpart who received standard error messages. Error messages presented in a programmer's first language were consistently preferred for their increased readability and comprehensibility, even when a quantitative reduction in overall debugging time was not observed.

Limitations

Following the discussion and conclusions drawn above, we highlight specific limitations that need to be recognized and taken into account for future studies.

The first limitation is the small sample size of the user study which limits the generalizability and validity of the results. The study only involved three participants who were randomly assigned to 1 of 3 groups. This means that the observed results may not reflect the views and behaviours of a larger population of novice programmers in South Africa.

The second limitation is the quality and quantity of the improved error messages. The improved error messages were created based on a limited set of user feedback. Out of the minimum of 56 first-year Computer Science tutors available on the Amathuba platform, Linked In, and WhatsApp where the participation survey was disseminated, the templates were primarily informed by the input of one person. Designing improved based on a single person's input and perception might not accurately represent the diverse range of users who could potentially encounter these errors. Moreover, although 62 templates were successfully designed, only 15 were tested in English and isiXhosa. To ascertain the broader utility and effectiveness of these error message templates, further testing and validation is necessary.

The last limitation in this study is related to the measurement of debugging time. The study's method of measuring debugging time focused solely on the time difference between a program's first successful execution and the first successful execution of the previously completed task. This approach did not account for other potentially influential factors, such as idle time when participants may have been distracted or the time they spent adapting to the new error messages. These unaccounted factors could have had an impact on the observed results and should be considered in future research to provide a more comprehensive understanding of debugging efficiency.

REFERENCES

[1] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 575–585.

[2] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[3] Brett A Becker, Kyle Goslin, and Graham Glanville. 2018. 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*. 640–645.

[4] Kees Van Deemter, Mariët Theune, and Emiel Krahmer. 2005. Real versus template-based natural language generation: A false opposition? *Computational linguistics* 31, 1 (2005), 15–24.

[5] Paul Denny, James Prather, and Brett A Becker. 2020. Error message readability and novice debugging performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 480–486.

[6] Paul Denny, James Prather, Brett A Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B Powell. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[7] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 55, 15 pages. https://doi.org/10.1145/3411764.3445696

[8] Saifaddin Galal. 2022. Statista. https://www.statista.com/statistics/1114302/distribution-of-languages-spoken-inside-and-outside-of-households-in-south-africa/. Accessed: March 21, 2023.

[9] Philip J Guo. 2018. Non-native english speakers learning computer programming: Barriers, desires, and design opportunities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–14.

[10] Raymond S Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students? Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 465–470.

[11] James Prather, Paul Denny, Brett A Becker, Robert Nix, Brent N Reeves, Arisoa S Randrianasolo, and Garrett Powell. 2023. First Steps Towards Predicting the Readability of Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 549–555.

[12] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 74–82.

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

[14] Adalbert Gerald Soosai Raj, Kasama Ketsuriyonk, Jignesh M Patel, and Richard Halverson. 2018. Does native language play a role in learning a programming language?. In *Proceedings of the 49th ACM technical symposium on computer science education*. 417–422.

[15] Emillie Thiselton and Christoph Treude. 2019. Enhancing Python Compiler Error Messages via Stack Overflow. *CoRR* abs/1906.11456 (2019). arXiv:1906.11456 http://arxiv.org/abs/1906.11456

Appendix A ERROR FREQUENCY TABLE

Table 9: Error messages ranked by frequency from Prichard's work [13].

Error Type	Error Message	Frequency
SyntaxError	invalid syntax	179624
NameError	name ” is not defined	97186
SyntaxError	unexpected EOF while parsing	26097
IndentationError	unindent does not match any outer indentation level	20758
SyntaxError	EOL while scanning string literal	16673
IndentationError	unexpected indent	16166
TabError	inconsistent use of tabs and spaces in indentation	13599
IndentationError	expected an indented block	11788
ValueError	invalid literal for func() with base #: ”	11592
TypeError	” object is not iterable	10327
TypeError	” object is not callable	9762
TypeError	can't multiply sequence by non-int of type ”	9656
TypeError	unsupported operand type(s) for +: ” and ”	8851
TypeError	unsupported operand type(s) for -: ” and ”	6843
IndexError	string index out of range	6707
SyntaxError	unexpected character after line continuation character	6326
TypeError	Can't convert ” object to str implicitly	6005
TypeError	unsupported operand type(s) for //: ” and ”	5874
AttributeError	” object has no attribute ”	5627
TypeError	” object is not subscriptable	5584
TypeError	string indices must be integers	5553
IndexError	list index out of range	5030
SyntaxError	can't assign to function call	4005
UnboundLocalError	local variable ” referenced before assignment	3360
SyntaxError	can't assign to operator	3277
ValueError	could not convert string to float: ”	2712
TypeError	unorderable types: func() > func()	2656
TypeError	not all arguments converted during string formatting	2643
TypeError	object of type ” has no func()	2633
TypeError	” object cannot be interpreted as an integer	2455
TypeError	list indices must be integers, not <TYPE>	2266
TypeError	an integer is required	2208
TypeError	unsupported operand type(s) for /: ” and ”	2080
TypeError	func() expected string of length #, but <TYPE> found	2026
RuntimeError	maximum recursion depth exceeded while calling a Python object	2025
TypeError	unsupported operand type(s) for ** or func(): ” and ”	1995
IndexError	list assignment index out of range	1734
TypeError	slice indices must be integers or None or have an __index__ method	1702
TypeError	unsupported operand type(s) for *: ” and ”	1658
SyntaxError	break outside loop	1475
TypeError	func() expected a character, but string of length # found	1453
SyntaxError	return outside function	1431
SyntaxError	can't assign to literal	1345
TypeError	unorderable types: func() < func()	1330
ValueError	? is not in list	1209
TypeError	func() argument must be a string or a number, not ”	1133
TypeError	can only concatenate list (not ”) to list	1090
TypeError	” object does not support item assignment	1070
ValueError	math domain error	1062
TypeError	func() takes exactly # arguments (# given)	912
RuntimeError	maximum recursion depth exceeded in comparison	867
TypeError	” is an invalid keyword argument for this function	832
TypeError	unorderable types: func() >= func()	806

Continued on next page

Table 9 – Continued from previous page

Error Type	Error Message	Frequency
SyntaxError	invalid character in identifier	784
TypeError	func() missing # required positional arguments: ”	749
TypeError	bad operand type for unary +: ”	648
ZeroDivisionError	integer division or modulo by zero	634
SyntaxError	continue not properly in loop	585
TypeError	unsupported operand type(s) for ˆ: ” and ”	480
TypeError	func() takes at least # arguments (# given)	426
TypeError	bad operand type for unary -: ”	403
RuntimeError	maximum recursion depth exceeded	373
TypeError	descriptor ” requires a ” object but received a ”	360
TypeError	func() takes no arguments (# given)	353
ValueError	list.remove(x): x not in list	346
ZeroDivisionError	division by zero	344
TypeError	a float is required	335
TypeError	func() argument must be a string or a number	323
SyntaxError	can use starred expression only as assignment target	310
TypeError	unorderable types: func() <= func()	284
TypeError	func() takes # positional arguments but # were given	261
TypeError	can only concatenate tuple (not ”) to tuple	248
TypeError	argument of type ” is not iterable	224
ImportError	No module named ”	210
TypeError	unsupported operand type(s) for %: ” and ”	203
SyntaxError	non-keyword arg after keyword arg	189
TypeError	func() argument after * must be a sequence, not <TYPE>	185
TypeError	unsupported operand type(s) for +=: ” and ”	185
TypeError	func() takes no keyword arguments	178
SyntaxError	EOF while scanning triple-quoted string literal	173
ValueError	substring not found	170
TypeError	func() expected # arguments, got #144	168
IndexError	pop index out of range	117
TypeError	func() takes at most # arguments (# given)	114
ValueError	func() arg is an empty sequence	107
TypeError	func() expected at most # arguments, got #	101
ValueError	func() arg not in range(0x#)	92
TypeError	unsupported operand type(s) for &: ” and ”	89
IndexError	tuple index out of range	87
TypeError	func() can’t convert non-string with explicit base	86
SyntaxWarning	name ” is assigned to before global declaration	84
FileNotFoundError	[Errno #] No such file or directory: ”	84

Appendix B ERROR VERSIONS

Table 10: Table depicting error versions from earlier Python version and current version used in the study

Older Version	New Version
SyntaxError: Invalid syntax	SyntaxError: expected ':'
SyntaxError: unexpected EOF while parsing	SyntaxError: '(' was never closed
SyntaxError: EOL while scanning string literal	SyntaxError: unterminated string literal (detected at line #)
SyntaxError: can't assign to operator	SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
SyntaxError: invalid character in identifier	SyntaxError: invalid character '‘' (U+208)
SyntaxError: can use starred expression only as assignment target	SyntaxError: can't use starred expression here
SyntaxError: non-keyword arg after keyword arg	SyntaxError: positional argument follows keyword argument
SyntaxError: EOF while scanning triple-quoted string literal	SyntaxError: unterminated triple-quoted string literal (detected at line 2)
TypeError: Can't convert '' object to str implicitly	TypeError: can only concatenate str (not " ") to str
TypeError: string indices must be integers	TypeError: indices must be integers or slices, not <>
TypeError: unorderable types: func() > func()	TypeError: '>' not supported between instances of 'function' and 'function'
TypeError: unorderable types: func() < func()	TypeError: '<' not supported between instances of 'function' and 'function'
TypeError: unorderable types: func() >= func()	TypeError: '>=' not supported between instances of 'function' and 'function'
TypeError: unorderable types: func() <= func()	TypeError: '<=' not supported between instances of 'function' and 'function'
TypeError: func() missing # required positional arguments: ''	TypeError: func() takes # positional arguments but # were given
TypeError: func() argument must be a string or a number, not ''	TypeError: func() argument must be a string, a bytes-like object or a real number, not ''
TypeError: descriptor '' requires a '' object but received a ''	TypeError: descriptor '' for '' objects doesn't apply to a '' object
TypeError: func() expected # arguments, got #	TypeError: func() takes # positional arguments but # were given
TypeError: func() takes at most # arguments (# given)	TypeError: func() takes # positional arguments but # were given
TypeError: func() takes exactly # arguments (# given)	TypeError: func() takes # positional arguments but # were given
TypeError: func() takes at least # arguments (# given)	TypeError: func() takes # positional arguments but # were given
TypeError: func() takes no arguments (# given)	TypeError: func() takes # positional arguments but # were given
TypeError: func() expected at most # arguments, got #	TypeError: func() takes # positional arguments but # were given
TypeError: func() argument after * must be a sequence, not <TYPE>	TypeError: Value after * must be an iterable, not int
TypeError: an integer is required	TypeError: '' object cannot be interpreted as an integer ''
TypeError: a float is required	TypeError: '' object cannot be interpreted as an integer ''
func() expected string of length #, but <TYPE> found	TypeError: '' object is not iterable
SyntaxWarning: name '' is assigned to before global declaration	SyntaxError: name 'x' is assigned to before global declaration
ImportError: No module named ''	ModuleNotFoundErrror: No module named ''
RuntimeError: maximum recursion depth exceeded while calling a Python object	RecursionError: maximum recursion depth exceeded while calling a Python object
RuntimeError: maximum recursion depth exceeded in comparison	RecursionError: maximum recursion depth exceeded in comparison
RuntimeError: maximum recursion depth exceeded	RecursionError: maximum recursion depth exceeded

Appendix C ERROR MESSAGES AND TEMPLATES

Table 11: Table of Error Templates and Messages. This table displays error templates and groups all the error messages they generate. If an error message does not conform to the structure of the error message template, it indicates that it changed. Refer to Table 10 above to see the corresponding change.

Template/Umbrella Template	Error Message(s)
[ERROR_TYPE]: expected '[VARIABLE]'	SyntaxError: invalid syntax
[ERROR_TYPE]: '[VARIABLE]' was never closed	SyntaxError: unexpected EOF while parsing
[ERROR_TYPE]: unterminated [TYPE] literal (detected at line [LINENUMBER])	SyntaxError: EOL while scanning string literal
[ERROR_TYPE]: unexpected character after line continuation character	SyntaxError: unexpected character after line continuation character
[ERROR_TYPE]: cannot assign to function call here.[SUGGESTION]	SyntaxError: can't assign to function call
[ERROR_TYPE]: cannot assign to expression here.[SUGGESTION]	SyntaxError: can't assign to operator
[ERROR_TYPE]: '[VARIABLE]' outside loop	SyntaxError: break outside loop
[ERROR_TYPE]: '[VARIABLE]' outside function	SyntaxError: return outside function
[ERROR_TYPE]: cannot assign to literal here.[SUGGESTION]	SyntaxError: can't assign to literal
[ERROR_TYPE]: invalid character '[VARIABLE]' (U+[NUMBER])	SyntaxError: invalid character in identifier
[ERROR_TYPE]: 'VARIABLE' not properly in loop	SyntaxError: continue not properly in loop
[ERROR_TYPE]: can't use starred expression here	SyntaxError: can use starred expression only as assignment target
[ERROR_TYPE]: positional argument follows keyword argument	SyntaxError: non-keyword arg after keyword arg
[ERROR_TYPE]: unterminated triple-quoted [TYPE] literal (detected at line [NUMBER])	SyntaxError: EOF while scanning triple-quoted string literal
[ERROR_TYPE]: name '[VARIABLE]' is assigned to before global declaration	SyntaxWarning: name '' is assigned to before global declaration
[ERROR_TYPE]: unindent does not match any outer indentation level	IndentationError: unindent does not match any outer indentation level
[ERROR_TYPE]: unexpected indent	IndentationError: unexpected indent
[ERROR_TYPE]: expected an indented block after '[VARIABLE]' statement on line [NUMBER]	IndentationError: expected an indented block after '' statement on line #
[ERROR_TYPE]: inconsistent use of tabs and spaces in indentation	TabError: inconsistent use of tabs and spaces in indentation
[ERROR_TYPE]: name '[VARIABLE]' is not defined	NameError: name '' is not defined
[ERROR_TYPE]: '[TYPE]' object is not iterable	TypeError: '' object is not iterable
[ERROR_TYPE]: '[TYPE]' object is not callable	TypeError: '' object is not callable
[ERROR_TYPE]: can't multiply sequence by non-int of type '[TYPE]'	TypeError: can't multiply sequence by non-int of type ''
[ERROR_TYPE]: unsupported operand type(s) for [OPERATOR]: '[TYPE]' and '[TYPE]'	TypeError: unsupported operand type(s) for +: '' and '' TypeError: unsupported operand type(s) for -: '' and '' TypeError: unsupported operand type(s) for //: '' and '' TypeError: unsupported operand type(s) for /: '' and '' TypeError: unsupported operand type(s) for **: '' and '' TypeError: unsupported operand type(s) for *: '' and '' TypeError: unsupported operand type(s) for ^: '' and '' TypeError: unsupported operand type(s) for %: '' and '' TypeError: unsupported operand type(s) for +=: '' and '' TypeError: unsupported operand type(s) for &: '' and ''
[ERROR_TYPE]: can only concatenate [TYPE] (not [TYPE]) to [TYPE]	TypeError: Can't convert '' object to str implicitly TypeError: can only concatenate list (not '') to list TypeError: can only concatenate tuple (not '') to tuple

[ERROR_TYPE]: '[TYPE]' object is not subscriptable	TypeError: '' object is not subscriptable
[ERROR_TYPE]: [TYPE] indices must be integers or slices, not [TYPE]	TypeError: string indices must be integers TypeError: list indices must be integers, not <TYPE>
[ERROR_TYPE]: [OPERATOR] not supported between instances of '[VARRIABLE]' and '[VARIABLE]'	TypeError: unorderable types: func() < func() TypeError: unorderable types: func() > func() TypeError: unorderable types: func() <= func() TypeError: unorderable types: func() >= func()
[ERROR_TYPE]: not all arguments converted during [TYPE] formatting	TypeError: not all arguments converted during string formatting
[ERROR_TYPE]: object of type '[TYPE]' has no [FUNCTION]	TypeError: object of type '' has no func()
[ERROR_TYPE]: '[TYPE]' object cannot be interpreted as an [TYPE]	TypeError: '' object cannot be interpreted as an integer TypeError: an integer is required TypeError: a float is required
[ERROR_TYPE]: [FUNCTION] expected a [Type], but [TYPE] of length [NUMBER] found	TypeError: func() expected a character, but string of length # found TypeError: func() expected string of length #, but <TYPE> found
[ERROR_TYPE]: slice indices must be integers or None or have an __index__ method	TypeError: slice indices must be integers or None or have an __index__ method
[ERROR_TYPE]: '[TYPE]' object does not support item assignment	TypeError: '' object does not support item assignment
[ERROR_TYPE]: '[VARIABLE]' is an invalid keyword argument for [FUNCTION]	TypeError: '' is an invalid keyword argument for this function
[ERROR_TYPE]: bad operand type for unary [OPERATOR]: '[TYPE]'	TypeError: bad operand type for unary +: '' TypeError: bad operand type for unary -: ''
[ERROR_TYPE]: descriptor '[VARIABLE]' for '[TYPE]' objects doesn't apply to a '[TYPE]' object	TypeError: descriptor '' requires a '' object but received a ''
[ERROR_TYPE]: [FUNCTION] argument must be a string, a bytes-like object or a real number, not '[TYPE]'	TypeError: func() argument must be a string or a number, not '' TypeError: func() argument must be a string or a number
[ERROR_TYPE]: [FUNCTION] takes [NUMBER] positional argument but [NUMBER] were given	TypeError: func() takes # positional arguments but # were given TypeError: func() expected # arguments, got # TypeError: func() takes at most # arguments (# given) TypeError: func() takes exactly # arguments (# given) TypeError: func() takes at least # arguments (# given) TypeError: func() takes no arguments (# given) TypeError: func() expected at most # arguments, got # TypeError: func() missing # required positional arguments: ''
[ERROR_TYPE]: Value after [OPERATOR] must be an iterable, not [TYPE]	TypeError: func() argument after * must be a sequence, not <TYPE>
[ERROR_TYPE]: [FUNCTION] takes no keyword arguments	TypeError: func() takes no keyword arguments
[ERROR_TYPE]: [FUNCTION] can't convert non-string with explicit base	TypeError: func() can't convert non-string with explicit base
[ERROR_TYPE]: invalid literal for [FUNCTION] with base [NUMBER]: '[VALUE]'	ValueError: invalid literal for func() with base #: ''
[ERROR_TYPE]: could not convert [TYPE] to [TYPE]: '[VALUE]'	ValueError: could not convert string to float: ''
[ERROR_TYPE]: [NUMBER] is not in [TYPE]	ValueError: ? is not in list
[ERROR_TYPE]: math domain error	ValueError: math domain error
[ERROR_TYPE]: [TYPE].remove(x): x not in [TYPE]	ValueError: list.remove(x): x not in list
[ERROR_TYPE]: substring not found	ValueError: substring not found
[ERROR_TYPE]: [FUNCTION] arg is an empty sequence	ValueError: func() arg is an empty sequence
[ERROR_TYPE]: [FUNCTION] arg not in range([NUMBER]x[NUMBER])	ValueError: func() arg not in range(0x#)

[ERROR_TYPE]: [TYPE] index out of range	IndexError: string index out of range IndexError: list index out of range IndexError: tuple index out of range
[ERROR_TYPE]: [TYPE] assignment index out of range	IndexError: list assignment index out of range
[ERROR_TYPE]: pop index out of range	IndexError: pop index out of range
[ERROR_TYPE]: '[TYPE]' object has no attribute '[VARIABLE]'	AttributeError: '' object has no attribute ''
[ERROR_TYPE]: local variable '[VARIABLE]' referenced before assignment	UnboundLocalError: local variable '' referenced before assignment
[ERROR_TYPE]: maximum recursion depth exceeded while calling a Python object	RuntimeError: maximum recursion depth exceeded while calling a Python object
[ERROR_TYPE]: maximum recursion depth exceeded in comparison	RuntimeError: maximum recursion depth exceeded in comparison
[ERROR_TYPE]: maximum recursion depth exceeded	RuntimeError: maximum recursion depth exceeded
[ERROR_TYPE]: [TYPE] division or modulo by zero	ZeroDivisionError: integer division or modulo by zero
[ERROR_TYPE]: division by zero	ZeroDivisionError: division by zero
[ERROR_TYPE]: No module named '[VARIABLE]'	ImportError: No module named ''
[ERROR_TYPE]: [Errno [NUMBER]] No such file or directory: '[VARIABLE]'	FileNotFoundError: [Errno #] No such file or directory: ''

Appendix D ERROR MESSAGE TEMPLATES (INCLUDING STANDARD, IMPROVED AND TRANSLATED VERSIONS)

https://uctcloud-my.sharepoint.com/:x:/g/personal/tnzman002_myuct_ac_za/ESRe2DHeynVOhJBh90O0hXoBdwdXYHraJKpNq_BJrtkIrg?e=TkpTDn

URL Description: Standard, Improved, and Translated Error message Templates. All Templates Used in User Study are Highlighted in Yellow