UNIVERSITY OF CALIFORNIA SAN DIEGO

PABLO and PYRITE: Helping Novices Debug Python Code Through Data-Driven Fault Localization and Repair

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Benjamin Cosman

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Philip Guo
Professor James Hollan
Professor Sorin Lerner
Professor Joseph Politz

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ANOVA - Analysis of variance

AST - Abstract Syntax Tree

BOAT - Bag of Abstracted Terms [SSC^{+}17]

CFG - Context-Free Grammar

SMT - Satisfiability Modulo Theories [BSST09]

ACKNOWLEDGEMENTS

I would like to thank Ranjit Jhala for being my supportive and patient advisor, Phil Guo for the use of the PythonTutor dataset, and the rest of my committee Jim Hollan, Sorin Lerner, and Joe Politz.

I also want to thank my PABLO and PYRITE collaborators Maddy, George, Leon, Yao-Yuan, Ranjit, Kamalika, and Wes; special shout-out to Maddy for all her work on the human studies.

Work adapted in this dissertation

Chapter 1 and Chapter 3 are adapted from “PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization” in the proceedings of the 2020 Technical Symposium on Computer Science Education (SIGCSE), by Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer [CES$^+$20], as well as from an earlier version that was submitted for publication to ESEC/FSE 2019.

Chapter 2 is adapted from the dissertation author’s Research Exam, entitled “Synthesis Techniques for CS Education” and presented in Spring 2017.

Chapter 4 describes unpublished work done in collaboration with Madeline Endres, Georgios Sakkas, Westley Weimer, and Ranjit Jhala.

The dissertation author was the primary investigator and author of these works.

VITA

2014	B.S. in Computer Science, California Institute of Technology
2017	M.S. in Computer Science, University of California San Diego
2021	Ph.D. in Computer Science, University of California San Diego

PUBLICATIONS

B. Cosman, M. Endres, G. Sakkas, L. Medvinsky, Y. Yang, R. Jhala, K. Chaudhuri, W. Weimer, "PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization," Technical Symposium on Computer Science Education (SIGCSE 2020)

G. Sakkas, M. Endres, B. Cosman, W. Weimer, R. Jhala, "Type Error Feedback via Analytic Program Repair," Conference on Programming Design and Implementation (PLDI 2020)

M. Endres, G. Sakkas, B. Cosman, R. Jhala, W. Weimer, "InFix: Automatically Repairing Novice Program Inputs," International Conference on Automated Software Engineering (ASE 2019)

B. Cosman and R. Jhala, "Local Refinement Typing," International Conference on Functional Programming (ICFP 2017)

P. Vekris, B. Cosman, R. Jhala, "Refinement Types for Typescript," Conf. on Programming Languages Design and Implementation (PLDI 2016)

P. Vekris, B. Cosman, R. Jhala, "Trust but Verify: Two Phase Typing for Dynamic Languages," European Conf. on Object Oriented Programming (ECOOP 2015)

ABSTRACT OF THE DISSERTATION

PABLO and PYRITE: Helping Novices Debug Python Code Through Data-Driven Fault Localization and Repair

by

Benjamin Cosman

Doctor of Philosophy in Computer Science

University of California San Diego, 2021

Professor Ranjit Jhala, Chair

As dynamically-typed languages grow in popularity, especially among beginning programmers, novices have an increased need for scalable, helpful feedback for fixing their bugs. Localization and repair can be ambiguous: not all repairs which prevent the program from crashing are equally useful for beginners. We propose scalable approaches for fault localization and repair for dynamic languages that are helpful for debugging and generalize to handle a wide variety of errors commonly faced by novice programmers. We base our approach on a combination of static, dynamic, and contextual features, guided by machine learning. We evaluate on over 980,000 diverse real user interactions across four years from the popular PythonTutor.com website, which is used both in classes and by non-traditional learners. We find that our approach is scalable, general, and quite accurate: up to 77% of these historical novice users would have been helped by our top-three localization responses, compared to 45% for the default interpreter, and we successfully synthesize repairs to 76% of our historical buggy programs. We also conducted two human studies. Participants preferred our localization approach to the baseline ($p = 0.018$), and found it additionally useful for bugs meriting multiple edits. Participants found our repairs to contain helpful information beyond the baseline in 45% of programs.

Chapter 1

Introduction

Many programmers of dynamically-typed languages, especially novices unfamiliar with debugging tools and lacking the expertise to interpret compiler error messages, can have a difficult time pinpointing (and then fixing) what *caused* their program to crash [PO11, ZM14, Chr14, PKW14a, CE14, NT03, SSW04]. Solutions to this problem are needed as class sizes grow and increasing numbers of non-traditional learners progress with limited access to human instructional support.

Dynamically-typed languages like Python are increasingly popular for teaching programming [Guo13], and many of those learning Python require help debugging their code. For example, the PythonTutor online debugging and visualization tool alone is visited by more than sixty thousand users per month, and almost half of the program crashes they face take multiple attempts to resolve. Localizing errors in Python is especially hard because there is no (static) type information, and because we seek to handle not just type errors but many other kinds of errors as well (IndexError, ValueError, etc.) which are manifestations of a wide variety of root failure causes.

One of the key difficulties in finding and fixing bugs is that there are often multiple logically-valid fixes — only one of which may be desired by the programmer [SSC$^+$17]. Consider

```
1 year = int(time.strftime("%Y"))
2 age = input("Enter your age")
3 print("You will be twice as old in:")
4 print(year + age)
```

Figure 1.1: A simple program with multiple fault localizations (lines 1 and 2).

the program in Figure 1.1, adapted from our dataset (Section 3.3.1). The program attempts to carry out some simple arithmetic based on a given number and the current year. When executed, however, the program raises an exception on line 4 (when adding an integer to a string). One possible "fix" would be to *remove* the `int()` conversion from line 1, in which case both variables hold string values at runtime, and the `+` on line 4 would be interpreted as string concatenation. However, another reasonable fix (and the one that the programmer actually used in this case) is to add an `int()` cast on line 2. While both fixes yield well-typed programs, the second is more likely to correspond to developer intent, and is thus more likely to be helpful as a localization and debugging aid.

Thus, a useful debugging aid needs to provide debugging hints that are *helpful*, implicating locations that correspond to developer intent or aid novice learning. It should also be *general*, applying to a wide variety of the errors that novices frequently encounter, including more complex bugs involving multiple simultaneous conceptual mistakes spanning multiple lines. Finally, it should be *scalable*, working quickly in large classes and non-traditional settings.

There is lots of prior work on giving debugging feedback for programmers. However, most of it does not meet our criteria in one or more one of these ways:

- It is unable to distinguish between multiple formally-implicated fault localizations or repairs. We want to choose repairs that are correct, but more importantly we want them to be *helpful* to novices.
- It makes extensive use of a static type system. We however are targeting Python which is dynamically-typed, and novices who may not have learned yet about types at all.

- It requires more from the programmer than just some buggy Python code, e.g. a test suite. We want to help even brand new Python programmers who may not be capable yet of writing tests.

- It requires a more traditional educational setting - either a large pool of other students solving the same problem, or the intervention of an instructor who can provide reference implementations, expert feedback, etc. We want to help even in the case of non-traditional learners.

In Chapter 2 we look at some of the related work on feedback for novice programmers in detail; additional related work appears in each chapter.

Our key insight is that, while it is very difficult to resolve the root cause of a crash by looking at a single program at a time (as the example above shows), we can *learn* a suite of heuristics for how to debug and fix programs from a large corpus of real-world examples of novices fixing their own bugs. Novice-written bugs and fixes contain static, dynamic and contextual information about where errors appear frequently *in practice* and about how those errors can be fixed.

In **PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization** (Chapter 3), we build on this insight to present a novel approach to localizing defects in unannotated, beginner-written programs without test suites. We present a set of static, dynamic, and contextual features of a user's program which are sufficient for a machine learning model to pinpoint faulty locations with high accuracy. We present both an automated evaluation on almost a million real user interactions from the PythonTutor.com website, and also a human study which confirms that our error localizations are indeed helpful and better than those provided by the default Python interpreter.

In **Pyrite: Analytic Program Repair for Python Novices** (Chapter 4), we extend this work by demonstrating how to repair the faulty terms implicated by PABLO. We train a second machine learning model so that we can predict not only error locations but also *fix templates* which describe how those locations could be repaired. We then use enumerative synthesis to turn these into concrete program repairs. We again provide both an automated evaluation and a user study to conclude our repairs are helpful.

Finally, Appendix A discusses an attempted but unsuccessful alternate method for generating repairs using adversarial examples, Appendix B goes into further details on the machine learning features used by PABLO and PYRITE, and Appendix C contains the stimuli used in our user studies.

Acknowledgements for Chapter 1

This chapter is adapted from the introduction of "PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization" in the proceedings of the 2020 Technical Symposium on Computer Science Education (SIGCSE), by Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer [CES$^+$20], as well as from an earlier version that was submitted for publication to ESEC/FSE 2019. The dissertation author was the primary investigator and author of this material.

Chapter 2

Related Work: Feedback for novices

We begin by surveying existing work which like ours provides feedback for novice programmers. (Other kinds of related work appear near the ends of the following chapters.) Providing feedback for novices involves a unique set of challenges and opportunities:

Challenges. Novice programmers are especially vulnerable to bad error messages since they have less experience interpreting them. For classical feedback of a location and an error message, it becomes especially important to get the location right because novices often ignore the message entirely in favor of manually inspecting the flagged location [JvdBvDH93]. Novice programmers may also write buggier code in general than is seen in the wild, so analyses must be prepared to handle an unusually high density of bugs. This is difficult because it is harder to handle more than one bug at once - for example if a method of fixing a bug relies on confirming afterward that the bug is fixed by running the code, then the presence of a second bug will mask the successful repair of the first bug.

Opportunities. The programs novices are writing are usually shorter than code in the wild, so computationally expensive techniques that might time out on long programs may become practical for novice code. Secondly, we can assume the existence of an instructor who can do things like writing a canonical solution for student submissions to be tested against, grading a few submissions accurately so our tools can learn how to grade the rest, or even guessing what kind of mistakes the students are likely to make so our tools can watch out for them. Finally, on a given assignment, we can expect a lot of repetition: a large group of students will find comparatively few different ways of being right and being wrong; machine learning and clustering techniques are especially good at taking advantage of such patterns. (PABLO (Chapter 3) and PYRITE (Chapter 4) choose to forgo some of these opportunities in order to reach non-traditional learners who may not have instructional support or peers working on the same assignments.)

2.1 Feedback requiring expert input

We are targeting non-traditional learners who may not have institutional support, such as self-directed visitors to the PythonTutor website [Guo13]. However many related projects seek to provide feedback to novice programmers in the context of a class where there is an instructor and the students are all working on the same problem. They can thus get this domain expert to help by providing a reference implementation or a list of common errors that their students make.

Autograder

AutoGrader [SGSL13a] repairs student code submissions. They first call upon the instructor to provide an “error model” - a list of rewrite rules corresponding to mistakes that the instructor expects many students will make. For example, an instructor expecting that students might make an off-by-one error could create the rewrite rule $v = n \to v = \{n+1, n-1\}$.

AutoGrader then translates the student code into an expanded version where each statement that has a rewrite rule in the error model is expanded into a set of options, chosen by a hole - for example, using the above rewrite rule, the single line `a = 3` would be rewritten to something like Figure 2.1.

In the next step, the synthesis engine chooses a branch by filling in the `??` holes. The

```
choice = ?? # to be filled in later by the synthesizer
if choice == OPTION_0:
  a = 3 # Original student code was correct
elif choice == OPTION_1:
  a = 2 # Off-by-one
  totalCost++
elif choice == OPTION_2:
  a = 4 # Off-by-one
  totalCost++
```

Figure 2.1: Roughly what AutoGrader expands the line `a = 3` to, given an error model suggesting off-by-one bugs.

synthesis algorithm is based on Sketch [SL08], which uses Counterexample Guided Inductive Synthesis. The synthesizer is run with the hard constraint that the program be correct and then the additional constraint that `totalCost` should be minimized. Since `totalCost` is increased in every case except the one corresponding to the student's original code, this constraint means the synthesized solution will retain as much of the original code as possible.

In evaluation, they determine that AutoGrader can correct 64% of student submissions on a variety of real-world benchmarks in a reasonable amount of time. Some of these corrections involve up to 4 simultaneous fixes.

CPSGrader

CPSGrader [JDJS14] is a tool for automatically grading (simulated) Cyber-Physical Systems like robot controllers. The example setting is a class where students design a controller for a robot in a simulated environment, which should be graded based on how well it completes tasks like navigating around obstacles and climbing hills. The instructor is expected to provide "parameterized tests", corresponding to properties like the robot successfully progressing at least x distance up the hill in t seconds. The instructor also needs to write some reference implementations, both positive and negative. CPSGrader then tries various concrete values of each test's parameters to figure out under which values the test successfully classifies the reference implementations.

This strategy would not work for arbitrary tests, since figuring out appropriate parameters could require an intractable exhaustive search. Instead, they focus on monotonic tests - ones where there is a single threshold value between passing and failing. For example, if a test checks that a robot never veers more than θ degrees off course, then any robot that fails for some value of θ will also fail for all smaller values, and any robot that passes will also pass for all larger values, so it suffices to find one threshold that separates passing from failing and test all submissions using only that one value. In the case of multiple parameters, monotonicity guarantees a frontier of threshold tests rather than a single value, but this is still much better than testing *every* possible assignment to the parameters. A parameterized test can be proven to be monotonic by an SMT solver using prior work [JDDS13]. In their evaluation, they demonstrate that CPSGrader runs in a reasonable amount of time and can correctly classify the vast majority of student submissions in their benchmark set.

In followup work, they propose improving CPSGrader using active learning and clustering techniques [JJDS15]. Instead of having the instructor evaluate a fixed set of implementations on every test, they group similar student submissions into clusters and then choose representatives for the instructor to look at. In keeping with their earlier approach of running simulations rather than analyzing any code directly, similarity between submissions is measured by how similarly the robots behave in simulation, as computed using the "dynamic time warping" distance measure which can control for similar behaviors occurring at different times or speeds [Gio09].

ITAP

The Intelligent Teaching Assistant for Programming (ITAP) [RK17] tries to come up with the smallest possible code change that leads the novice in the right direction (see Figure 2.2). They have three goals for their hint:

- The hint code should compile even if used unchanged. In their target language of

```python
# A known solution:
...
def isGroundWet():
  return (isRaining and isOutdoors)
```

```python
# A student submission:
...
def isGroundWet():
  if isRaining:
    return True
  else:
    return False
```

```python
# ITAP's hint:
...
def isGroundWet():
  if isRaining and '[insert code here]':
    return True
  else:
    return False
```

Figure 2.2: ITAP providing a hint. Note that canonicalization allows it to ignore stylistic differences between the solution and the submission, like the use of an explicit if statement.

Python this is not very difficult since there isn't static type checking, so filler strings like `'[insert code here]'` can be used in place of any expression.

- The hint should be in the right direction, i.e. it should be the first step in a minimal path that will eventually lead to correct code.
- The hint should not *appear* to lead in the wrong direction, e.g. by decreasing the number of tests the code passes.

The existence of a test suite and at least one correct reference implementation is assumed; the test suite is the final arbiter of correctness.

To achieve this, they first find the closest known correct solution to the student's current state. However, it may be the case that not all edits required to get from one to the other are actually required for correctness. Thus they enumerate every subset of those edits, and check correctness of the resulting code for each. (This step is exponential-time, so it only works if there are a small number of edits; ITAP skips it with 16+.) ITAP is built to learn through use: states are stored as they are discovered through this process or by students. States are given a score representing both how good they actually are (in terms of distance from a correct solution and score on the test suite), and also how often students actually reach that state, since it is assumed that code other students have submitted in the past is more likely to be understandable to students even if it isn't correct.

Since the state space is enormous, a key part of the implementation is to reduce the state space by *canonicalizing* student code. This process includes choosing canonical variable names as well as other transformations that don't change semantics. They also need a *reification* procedure so that the hints they come up with for the transformed code can actually be applied to the original version. The authors' discuss their process further in [RK12]; similar procedures are present in several of the other algorithms surveyed here, including HelpMeOut below [HMBK10].

They evaluated ITAP on a collection of student submissions and found that it could successfully generate a chain of hints for the vast majority of them.

HelpMeOut

HelpMeOut [HMBK10] suggests bug fixes based on other users who had the same problem. It matches compile errors by canonicalized error text, and runtime errors by length of common stack trace. Users can vote up/down suggested fixes, and experts can annotate fixes in the database with explanations.

In a small user study of novice programmers, they found that the average user queried their tool once every 10 minutes and around half the queries returned a useful suggestion.

LAURA

LAURA [AL80] attempts to match a student submission to a reference implementation. It normalizes both into a canonical graph representation, then heuristically applies program transformations to try to make the two graphs equivalent, like constant propagation or reordering independent instructions. When it is able to match the graphs exactly it returns that the student code is correct (so there are never false positives). Otherwise, it relaxes its criteria for equivalence to allow similar but non-equivalent nodes to be identified; these differences are reported as potential bugs. This method is limited since many correct programs may be too different from the reference implementation for their algorithm to find the correspondence (or a correspondence might not even exist).

Feedback for Dynamic Programming

Kaleeswaran et al. focus on programming assignments that can be solved by dynamic programming [KSKG16]. They first cluster student submissions by various domain-specific static properties, like the number of nested loops used, which loop indicies are updated in which directions, etc. Then the instructor is asked to designate one submission in each cluster as correct, or write a correct submission for that cluster if none exists. Each submission is then compared to the closest correct representative, and checked for equivalence by an SMT solver. Since the clusters were chosen such that the programs being compared have the same structure, they can compare statements for equivalence basically line by line. If the programs are equivalent, they can safely identify the submission as also correct. Otherwise, they can generate as feedback the differences between the non-equivalent lines.

On their benchmarks which together contained over 2000 submissions, they found that each cluster, corresponding to one solution strategy, contained an average of 20 submissions. Most clusters had at least one correct submission (and this submission could usually be identified automatically using a few tests), so an expert just had to manually verify ~100 submissions and manually write 7 (for the clusters with no easily-identifiable correct submission) instead of looking at all 2000. The tool could then verify half of the submissions to be correct, and automatically generate feedback for 70% of the remainder.

One weakness of this method is how tailored it is to dynamic programming - since submission strategies are clustered using domain-specific features, it is not clear how this strategy could be generalized to a broader range of programming assignments. Additionally, the line-by-line equivalence comparison is too stringent: it is possible for a correct submission to differ enough from the manually-verified representative of its cluster that the tool will still mark it as incorrect and generate feedback on how to make it more similar to the representative.

2.2 Feedback using many submissions to the same assignment

Work in this section relies on a weaker assumption than in the previous - they still require all students to be working on the same assignment, but they no longer require an instructor to create additional resources like reference implementations. This is thus closer to our desired setting, though it is still not directly applicable since we want our system to work even when we don't know what the novice is working on and may not have other examples of novices solving that specific problem.

Refazer (and related work)

Refazer [RSD$^+$17] takes as input a codebase that has similar edits performed multiple times, and learns those code transformations so it can apply them elsewhere in the same codebase. The authors identify two domains where this type of codebase is common:

1. Student assignments: Many students will make similar mistakes and fix them in similar ways; learning these fixes can help other students who make those mistakes.
2. Repetitive refactorings: In large codebases that are being refactored, developers often need to make the same change in many places (e.g. changing the name of some field); learning these changes can help find other locations the same fixes should have been applied.

Refazer works in 3 steps: 1) separate out individual edits - basically connected modified AST nodes. 2) cluster the edits by edit distance (proxy for similarity). 3) synthesize rules for each cluster, favoring ones that use context (avoid false positives) but not too much context (avoid overfitting), and retain some of the modified code instead of replacing it with a constant snippet.

For their evaluation, they consider a corpus of student submissions, in which each student eventually submitted a correct version but also submitted at least one incorrect one. They determine that Refazer could have fixed some incorrect submission for 87% of the students. One flaw in this design is that they do not consider students who never successfully solved the problem at all: these students are probably most in need of help but may have buggier code that Refazer would do less well with. They also evaluate Refazer on various large open-source codebases; it does a good job of learning the repetitive edits in question. However, transformations learned from one programming assignment or one codebase are not found to be useful in a different assignment or codebase.

MistakeBrowser [HGS+17a] builds upon this work by using Refazer's clusters as a vehicle for an expert to provide hints: an instructor is presented with a cluster of student submissions that Refazer has determined can each be fixed by the same transformation, and the instructor can provide feedback to all students in that cluster. Other work that attempts to automate repeated edits in a single codebase includes LASE [MKM13] and those authors' prior work SYDIT [MKM11] which attempts to do the same thing based on a single edit (albeit with less success).

Learning program embeddings

Piech et al. use machine learning techniques to categorize programs by their functionality [PHN+15]. They target programming environments like Karel, a language for education where programmers instruct a robot to move around a fixed grid and interact with objects [Pat81]. Karel does not allow the programmer to create their own variables, which means that there is a fixed natural representation of any program's memory: all one needs to store is the locations of objects in the grid without worrying about additional programmer-defined state.

They first gather data to learn from by running each program on various test inputs.

They record not only the output of the program as a whole, but also the inputs and outputs to various components of the program like the body of each while loop. Given a set of student submissions, their goal is to learn a model which assigns a matrix to each program representing some linear transformation from input state to output state. Since programs are not necessarily linear with respect to the original representation, they simultaneously learn a non-linear encoding and decoding scheme from memory states to vectors such that in the new domain, programs do perform linear (or almost linear) transformations. In order to distinguish between programs that are functionally equivalent (or almost so) but are implemented differently, they also learn matrices for smaller components of each program, e.g. the body of each while loop.

They determine empirically that the models they learn are reasonable. In particular, linear transformations should compose - given two matrices that accurately represent two programs, the product of those matrices should represent the programs run one after another, and they discover that when they multiply matrices like this their predictive power goes down only very slightly. Overall, they achieve an accuracy around 90% and a recall ranging from 10-50% depending on the assignment.

Finally they use the embeddings they have learned and 500 annotated programs to learn a mapping from programs to what feedback they should be given. As usual there is a tradeoff they are able to make between false positives and false negatives; when they fix precision at 90% they can get “force multiplication factors” of between 12 and 214 (meaning that each annotated program allows them to automatically grade between a dozen and a few hundred others) depending on how big their test set is and how complicated the assignment is.

2.3 Other kinds of feedback

We conclude this chapter with a few projects that produce more unusual kinds of feedback for novices.

NanoMaLy

The goal of NanoMaLy [SJW16] is to synthesize a dynamic witness for a static type error - it allows the user to explore the whole stack trace starting from a concrete input and ending with a term that can be evaluated no further, like `0 + []`. It does this by symbolic execution - the input is treated as a "hole" of unknown type, which is then refined whenever execution reaches a primitive operation that requires a particular type. They choose types so as to have the program crash as late as possible, and prove that the traces they find are "general" in that if this algorithm finds a way for a function to crash, then there is no type the function could possibly be assigned such that it *wouldn't* crash on some input of that type.

To make the execution trace as useful as possible, NanoMaLy provides ways to interact with it. The user can zoom in and out on the trace, viewing every small step in the evaluation or just a few. The user can also pick a specific term anywhere in the computation and view the evaluation trace for just that term.

They evaluate NanoMaLy by providing students in a final exam setting some buggy code and then either OCaml's error message or NanoMaLy's execution trace. More students were able to correctly identify and explain the bug when given NanoMaLy's trace.

DFAs

Finally, we venture beyond conventional programming to consider feedback for the related problem of constructing automata. Deterministic Finite Automata (DFAs) are essentially simple programs distilled down to flow charts: the nodes in the chart represent the program counter and a limited amount of memory, and the arrows between them handle control flow based on the input. A theory class that includes constructing automata is part of a standard undergraduate Computer Science curriculum.

In [ADG$^+$13], the authors propose a scheme to classify student errors in DFAs and, in a later work [DKA+15], to provide appropriate feedback. The types of errors they identify are

1. Problem syntactic: the student misunderstood a part of the problem description and built a DFA to a conceptually similar specification, for example, confusing "at least" ($\geq$) with "more than" ($>$).

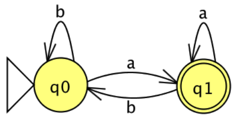
2. Problem semantic: the student failed to consider or correctly classify certain sets of strings - for example, they may understand that the problem is about strings of even length but not realize that the length of the empty string is also even.

3. Solution syntactic: the student probably has the right idea but made some errors in execution, e.g. by missing a couple transitions or accept states.

Each type of error has a different scheme for identification and elicits different feedback.

Problem syntactic The goal here is to provide a hint to the student by showing them what specification they actually satisfied so they can see the difference between that and the intended one. To do this, the authors created a specification language called MOSEL, a logic with the same expressive power as monadic-second order logic, which in turn is known to have the same expressive power as regular expressions and DFAs [Bü60]. The advantage of MOSEL is that it is designed to closely mimic how regular languages tend to be described in natural language. This facilitates the conversion between MOSEL and English descriptions of languages, and also means that a small edit distance between MOSEL specifications corresponds to a small distance between English specifications, and hence two languages which may more easily get confused with each other.

The challenge in identifying problem syntactic errors is that the direct translation from DFAs to MOSEL does not produce small or intuitive formulas, so that process is not useful for comparing the student's DFA to the correct one. Instead, they use brute force to enumerate small MOSEL formulas, and then test each for equivalence with the student's. The formula with



(a) A solution with a "problem semantic" error.

```
Your DFA is incorrect on
the following set of strings:
{s | s begins with a}
```

(b) A generalized counterexample

Figure 2.3: Feedback on a DFA which is supposed to accept strings that start and end with different letters.

the smallest edit distance to a canonical correct specification is returned to the student, after translation into natural language.

Problem semantic The goal here is to provide as a counterexample a concrete string or set of strings that the proposed DFA misclassifies. In the ideal case they provide a description of the entire set of strings that are misclassified (see Figure 2.3). The technique for finding this description is basically the same as in the problem syntactic case, since given two DFAs it is easy to construct another that accepts exactly those strings accepted by one but not the other, i.e. the strings that are misclassified. As above, sometimes no short description exists, in which case finding a long one could take a long time and might not be very useful anyway, so instead they come up with a description of a subset. If this too is impractical, they fall back on the well-known algorithm to find the shortest single string that is misclassified.

Solution syntactic The goal here is to find a small set of edits that can fix the solution, and then direct the student to which part of the DFA needs to be changed. They enumerate all small changes to the DFA, checking each for correctness. If they find a correct DFA in this way, the feedback for the student tells them roughly where to look, e.g. to check the transitions out of a particular state, or to check which states are accepting.

In a user study, these different kinds of feedback were appreciated and seemed to improve student performance over the baseline of just marking DFAs right or wrong, but not necessarily over simply providing short counterexample strings every time.

Acknowledgements for Chapter 2

This chapter is adapted from the dissertation author's Research Exam, entitled "Synthesis Techniques for CS Education" and presented in Spring 2017. The dissertation author was the sole author of this material.

Chapter 3

PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization

3.1 Introduction

Localizing the root cause of a failure is a key step in programming. With experience, developers pick up various tools to help with this process: for example, debuggers can be used to navigate failure traces, and in statically-typed langauges, type-checking can pinpoint classes of errors at compile time.

In this chapter, we present a novel approach to localizing defects in unannotated, beginner-written programs without test suites. We learn from a corpus of novice programs and produce answers which agree with human actions and judgments. Specifically, our system PABLO (Program Analytics-Based Localization Oracle):

1. takes as input a set of pairs of programs, each representing a program that crashes and the fixed version of that program

2. computes a *bag of abstracted terms* (BOAT) [SSC+17] representation for each crashing program: each AST node is represented by a vector of syntactic, dynamic, and contextual features

3. trains a classifier on these vectors

Then, given a new crashing program, PABLO will

1. compute BOAT vectors for each AST node of the program

2. classify each vector to obtain the likelihood that each corresponding node is to blame for the crash

3. return a list of *k* program locations, ranked by likelihood

Our hypothesis is that the combination of a rich corpus and domain-specific features admits automatically classifying common classes of defects and causes, thereby helping novices find the true sources of their bugs.

Our work is inspired by the NATE system [SSC+17] which introduced the notion of training classifiers over pairs of buggy-and-fixed programs. However, this work was limited to purely functional Ocaml programs where the static type discipline was crucial in both restricting the class of errors, and then providing the features that enabled learning. Furthermore, the work was only evaluated on a small corpus of programs comprising implementations of 20 different programming problems, leaving open the question of whether learning could accomodate a more *diverse* data set, and could ultimately *generalize* to different kinds of programs.

In this chapter, we show how learning to blame can scale up to Python by making the following concrete contributions:

1. First, we show how to identify a set of candidate features that allow us to train precise classifiers for dynamic imperative languages like Python. Specifically, we introduce and

evaluate static, dynamic, contextual and slice-based features that enable *scalable* data-driven localization for Python. PABLO systematically evaluates our feature set on beginner-written Python fragments. We find such features are all critical to our model, but that our model is not sensitive to error type and performs similarly across defect classes.

2. We evaluate PABLO in a user study and find that subjects find it *helpful*, and also *general* enough to provide useful debugging hints when multiple fixes are appropriate, which the baseline Python interpreter cannot do.

3. We perform a systematic evaluation on over 980,000 programs from four years of interactions with the PythonTutor [Guo13] website, a data set two orders of magnitude larger and significantly more diverse than that of similar prior work. PABLO is *helpful*, correctly identifying the exact expression that humans agree should be changed 59–77% of the time, compared to the Python interpreter's baseline of 45%, and *general*, retaining strong accuracy in all of the most common classes of bugs that novices encounter.

Note that the term *fault localization* is used differently in different contexts, from dynamic approaches like Tarantula [JH05] to static analyses like Mycroft [LCSS16] based on semantic information. These approaches are also evaluated on a diverse set of criteria, from producing a minimal correcting set [LCSS16] to agreeing with human intuition [SSC+17]. Our approach, PABLO, is a dynamic approach intended to agree with human intuition in a general setting where we do not assume we have type annotations or tests. As discussed further in Section 3.4, PABLO is thus not directly comparable to spectrum-based methods that require a test suite [JH05, CKF+02, AZG06, XM14, BLLLGG16, LZ17, SY17], systems that use static type information to make theoretical optimality guarantees [ZM14, LCSS16], or attempts to retrofit static type systems into dynamic languages, such as TypeScript [BAT14, VCJ16, FM14] for JavaScript and various gradual and static type approaches for Python [SPvR+, VKSB14] or Ruby [FAFH09, ACFH11].

3.2 Algorithm Overview

We present PABLO, an algorithm for accurately localizing faults [JH05] in dynamically-typed, beginner-written Python programs that exhibit non-trivial uncaught runtime exceptions. We do not consider syntax errors or references to undefined variables. Our algorithm uses machine learning models based on static, dynamic, contextual and slicing features to implicate suspicious expressions. Unlike short ranked lists [PO11, Sec. 5.6], voluminous fault localization output is not useful to developers in general [WPO15] and novices in particular [Koh19]. We thus produce Top-1 and Top-3 rankings; short lists are especially relevant for novices, who frequently make mistakes spanning multiple program locations.

Our algorithm first extracts static and dynamic features from a Python program (Section 3.2.2). Next, using a labeled training corpus, we train a machine learning model over those features (Section 3.2.4). Once the model has been trained, we localize faults in new Python programs by extracting their features and applying the model.

Drawing inspiration from localization algorithms such as NATE [SSC$^+$17] and the natural language processing term frequency vector (or "bag of words") model [SM86], we represent each buggy program as a "bag of abstracted terms". A *term* is either a statement or expression.

3.2.1 Model Feature Intuition and Extraction

We observe that many errors admit multiple logically-valid resolutions (see Figure 1.1): we thus cannot effectively localize through type constraints alone. Instead, we use static features to capture structured program meaning, contextual features to capture the relationship between a program fragment and its environment, and dynamic features to reason about conditional behavior.

Dynamic features are calculated using a trace of the program [BDB] after applying a semantics-preserving transformation [FSDF93] that admits expression-level granularity (instead of Python's whole lines).

3.2.2 Model Features

We provide an overview of our model features below; further details are in Appendix B.

Syntactic Forms (Static)

We hypothesize that certain syntactic categories of terms may be more prone to bugs than others, especially for beginner programmers. For example, students might have more trouble with loop conditions than with simple assignments. Thus the first feature we consider is the syntactic category of a node. This feature is categorical, using syntax tree labels such as Return or Import for statements and Variable or Application for expressions.

Expression Size (Static)

This numeric feature counts the number of descendents in each subtree. Our intuition is that larger, complex expressions may be more fault prone.

Type (Dynamic)

We observe that some types may be inherently suspect, especially for beginner-written code. For example, there are few reasons to have a variable of type **`NoneType`** in Python. This categorical feature includes all basic Python types (**`int`**, **`tuple`**, etc.) and three special values: **`Statement`** for statements, **`Unknown`** for expressions that are never evaluated, and **`Multiple`** for expressions which are evaluated multiple times in a trace and change type.

Slice (Dynamic)

The goal of this feature, roughly equivalent in purpose to the type error slice in NATE [SSC+17], is to help eliminate terms that cannot be the source of the crash. We compute a dynamic program slice [KL88]: a set of terms that contributed at runtime to the observed exception. This boolean feature encodes whether the term is a member of the slice. Our slicing algorithm is discussed further in Section 3.2.3.

Crash Location (Dynamic)

We observe that the precise term raising the exception is frequently useful for understanding and fixing the bug. This boolean feature flags the exception source.

Exception Type (Dynamic)

The type of error thrown is useful for localization. For example, every division term may be more suspicious if the uncaught exception is `DivisionByZero`. We encode the exception type categorically.

Local contextual features

Our BOAT representation, like term frequency vectors in general, does not include contextual information such as ordering. However, the meaning of an expression may depend on its context.

As an example, consider the `0` terms in both `x = 0` and `x / 0`. Both instances of `0` have the same syntactic form, size, type, etc. We may prefer to implicate the `0` in `x / 0` as suspicious, especially for beginner-written programs, but cannot distinguish it without surrounding contextual information. Structures like abstract syntax trees and control flow graphs capture such contextual information, but are not immediately applicable to machine learning.

We desire to encode such information while retaining the use of scalable, accurate, off-the-shelf machine learning algorithms that operate on feature vectors. We thus embed contextual information in a vector, borrowing insights from standard approaches in machine learning. We associate with each term additional features that correspond to the static and dynamic features of its parent and child nodes. For representational regularity, we always model three children; terms without parents or three children are given special values (e.g., zero or NotApplicable) for those contextual features.

Global contextual features

We hypothesize that advancing novices will both use a wider range of Python syntax and face different kinds of errors. We thus add boolean features indicating which node types appear anywhere in the program. These features will be sparsely populated for all nodes in simpler programs, and densely populated in programs using richer language features.

3.2.3 Dynamic Slicing Algorithm

Slicing information can help our model avoid implicating irrelevant nodes in the fault localization. Program slicing is a well-studied field with many explored tradeoffs (e.g., see Xu *et al.* for a survey [XQZ$^+$05]). We desire a slicing algorithm that can be computed efficiently (to scale to hundreds of thousands of instances) yet will admit high accuracy: we achieve this by focusing on features relevant to beginner-written programs. We follow the basic approach of Korel and Laski [KL88, KL90], building a graph of data and control dependencies. We then traverse the graph backwards, starting at the execution step where the error occurred, to collect the set of terms that the excepting line transitively depended on. This excludes lines that could not have caused the exception, such as lines that never ran, or lines that had no connection to the step where the exception happened.

Figure 3.1 is an example of the input and output of the slicing algorithm. The slicer takes as input a python source string and a list of inputs, to be fed into the program whenever `input()` is invoked. The slicer outputs the program elements included in the slice.

The slicer proceeds by creating an execution trace of the example program. At each execution step the trace includes the line number being run and the state of the heap and variables. In the example in Figure 3.1, lines $1, 2, 3, 4$ and 8 are executed. It then iterates forward through

```
1  x = input() # 42
2  if x < 20:
3    y = 5
4    z = 123
5  else:
6    y = 0
7    z = 321
8  assert(y != 0)
```

(a) Pre-Slicing

```
1  x = 42
2  if x < 20:
3
4
5  else:
6    y = 0
7
8  assert(y != 0)
```

(b) Post-Slicing

Figure 3.1: A program being sliced (left) and the resulting slice (right) with respect to the assertion on line 8.

```
1  x = input() # The user inputs 0
2  if x != 0:
3    x = 1
4  print("One over your number is: %d" % (1 / x))
```

Figure 3.2: In this program, the programmer intended the placeholder assignment (`x = 1`) to be used in case the user inputs `0`. The defect is that the incorrect condition (`x != 0` rather than `x == 0`) was employed. However, the condition expression will not be included in the slice (which would include lines 1 and 4), because the presence of the conditional does not affect the behavior of the program.

the trace, recording which execution steps have a define-use relation, and which execution steps have a test-control relation. The define-use relation is between a step where the value of a variable is used, and the step where that variable was last defined. In Figure 3.1, a define-use relationship exists between lines 1 and 2 and between 3 and 8. A test-control relation is between the execution of a control flow statement and any statements that were run as a result of it. In Figure 3.1, a test-control relationship exists between the if statement at line 2 and the assignments on lines 3 and 4. After building up these relations, the slicer starts at the execution step causing an exception (the execution of line 8), and builds the exception's set of dependencies by iteratively adding to the set any step that has a define-use or test-control relationship with a step already in the set. Finally, the algorithm maps all of the execution steps to their corresponding program element locations and returns the result.

Dynamic slicing involves an unavoidable tradeoff between unsoundness and over-

```
1 while true:
2   x = input()
3   if x != 0:
4     break
5   print("One over your number is: %d" % (1 / x))
```

Figure 3.3: In this example, the programmer attempts to handle invalid input by exiting the loop. Our slicer records that line 5 has a control dependency on the condition on line 3, correctly placing the buggy condition in the program slice.

approximation. For example, when the condition of an `if` statement is not met and so the code it guards is not run, we exclude the entire `if` block from the slice, choosing unsoundness (because the bug may indeed be in the `if` condition) instead of over-approximation (for example, including *every* control-flow condition in the slice). Thus if the bug is in the condition, our dynamic slice will miss it because the very defect we are trying to localize causes the conditional statement and its body to become irrelevant (see Figure 3.2). We observe that one common case this strategy fails is the "early break" case (Figure 3.3). We thus check if a break, return, or other statement for escaping structured control flow is present inside of a conditional statement, and add dependencies in the dependency graph between the enclosing conditional and the statements that would have been skipped by the break or return. While this heuristic is effective in practice, it does not overcome all related problems: we thus treat slice information as one of many features rather than as a hard constraint.

3.2.4 Machine Learning Model Generation

To apply machine learning, we first formulate the problem as a standard binary classification problem. For each term in a buggy program, we extract the features described in Section 3.2.2, and we assign it a label representing whether it should be blamed (see Section 3.3.2). We represent all features numerically by performing one-hot encoding on categorical features.

We choose to work with random forest models [Bre01], which train groups of decision trees, to balance accuracy with scalability to our large dataset. Each decision tree is a binary tree of simple thresholding classifiers. At each node, the training procedure chooses a feature, and then directs each incoming sample to one of its two children by comparing that feature to the chosen threshold. The feature and threshold are chosen to minimize the impurity of the two resulting partitions (e.g., measured by the Gini index [Bre17] or entropy [Qui86]). To mitigate overfitting, this process stops when the number of samples reaching a node is too small or the tree has grown too deep. When a test sample reaches a leaf of the tree, the model's prediction is based on a majority vote of the labels of the training samples that reached that leaf, with confidence determined by the proportion of training samples that agree with it. Decision trees scale well to large datasets and can learn non-linear prediction rules [Qui14, Qui86, Bre17].

Decision trees are prone to overfitting. To mitigate this problem, each tree in a random forest is trained on a subset of the data and a subset of the features. The prediction and confidence of the model as a whole is a weighted average of the predictions and confidences of the individual trees. Random forests thus trade some of the low computational cost of a plain decision tree for additional accuracy. We use 500 trees, each with a maximum depth of 30. Other parameters use the default SCIKIT-LEARN [PVG+11] settings.

Training methodology Given feature vectors describing every term of every program in our dataset, we train a model on a random 80% of the data and report the model's performance on the remaining 20%. To avoid inappropriate duplication between testing and training data, programs by the same user are always assigned together to either training or testing. We report the average of five such 80–20 splits. Each trained model takes in a feature vector representing a single term in a buggy program, and returns a confidence score representing how likely it is that the term was one of the terms changed between the fixed and buggy programs. We treat the model as providing a *ranking* over all terms by confidence. For a given *k*, we score the model based on Top-*k* accuracy: the proportion of programs for which a correct answer (i.e., a term that was actually changed historically) is present in the top *k* results. This is an imbalanced dataset in that non-buggy terms are much more common than buggy terms, so during training we re-weight to the reciprocal of the frequency of the corresponding class.

3.3 Evaluation

We conducted both a large-scale empirical evaluation of PABLO and also a human study to address these research questions:

RQ1 Do our localizations agree with human judgements?

RQ2 Which model features are the most important?

RQ3 How well does our algorithm handle different Python errors?

RQ4 Is our algorithm accurate on diverse programs?

RQ5 Do humans find our algorithm useful when multiple lines need to be edited?

3.3.1 Dataset and Program Collection

Our raw data consist of every Python 3 program that a user executed on PythonTutor.com [Guo13] (not in "live" mode) from late 2015 to end of 2018, other than those with syntax errors or undefined variables. Each program which crashes (throws an uncaught Python exception) is paired with the next program by the same user that does not crash, under the assumption that the latter is the fixed version of the former. We discard pairs where the difference between crashing and fixed versions is too high (more than a standard deviation above average), since these are the most likely to be violations of that assumption (e.g. the program that does not crash is an unrelated submission or a complete refactoring, rather than a bug fix). We also discard submissions that violate PythonTutor's policies (e.g., those using forbidden libraries).

To balance ease of prototype implementation against coverage for beginner-written programs, we also restrict the dataset to programs written in a simpler subset of Python, excluding programs that use the following more complicated features:

- Assignments where the left hand side is not a variable or a simple chain of attribute indexing or subscripting
- Assignments where attribute indexing or subscripting on the left hand side means something other than the default, (e.g., if the operations are overridden by a class)
- Lambda, generator and starred expressions
- Set and dictionary comprehensions
- Await, yield, and yield from
- Variable argument ellipsis
- Coroutine definitions and asynchronous loops
- Delete, with and raise statements

Ultimately, the dataset used in this evaluation contains 985,780 usable program pairs, representing students from dozens of universities (PythonTutor has been used in many introductory courses [Guo13]) as well as non-traditional novices.

3.3.2 Labeled Training and Ground Truth

Our algorithm is based on supervised machine learning and thus requires labeled training instances — a ground truth notion of which terms correspond to correct fault localizations. We use the terms changed in fixes by actual users as that ground truth. Many PythonTutor interactions are iterative: users start out by writing a program that crashes, and then edit it until it no longer crashes. Our dataset contains only those crashing programs for which the same user later submitted a program that did not crash. We compute a tree-diff [LLL09] between the original, buggy submission and the first fixed submission. For example, if the expression `len({3,4})` is changed to `len([3,4])`, then the node in the tree corresponding to the set `{3,4}` as a whole will appear in the diff (since it has been changed to a list), but neither its parent node (the enclosing `len` call) nor its children nodes (the integer literals) will appear since they were not changed.

We define the *ground truth* correct answer to be the set of terms in the crashing program that also appear in the diff. We discuss the implications of this choice in Section 3.3.9.

Given that notion of ground truth, a candidate fault localization answer is *accurate* if it is contained in the ground truth set. That is, if the human user changed terms X and Y, a technique (either our algorithm or a baseline) is given credit for returning either X or Y. A ranked response list is *top-k accurate* if any one of the top k answers is in the ground truth set.

For the overall accuracy experiments (Section 3.3.3), we employ a random forest as our machine learning algorithm, finding it to provide the best accuracy on our dataset with a scalable training and testing time. For the remaining experiments we use decision trees for increased training speed.

3.3.3 RQ 1 — Fault Localization Accuracy

We train random forests and compute their Top-1, Top-2, and Top-3 accuracy. For a baseline we compare to the standard Python interpreter, i.e., blaming the expression whose evaluation raises the uncaught exception. For fairness, we modify the Python interpreter to report expressions instead of its default of whole lines (see Section 3.2.1). We discuss other fault localization approaches and why they are not applicable baselines for our setting in Section 3.4.

As shown in Figure 3.4, PABLO produces a correct answer in the Top-1, Top-2, and Top-3 rankings 59%, 70%, and 77% of the time (to two significant figures). The expression blamed by the Python interpreter is only changed by the user 45% of the time.

Thus, our most directly-comparable model (Top-1), significantly outperforms this baseline.

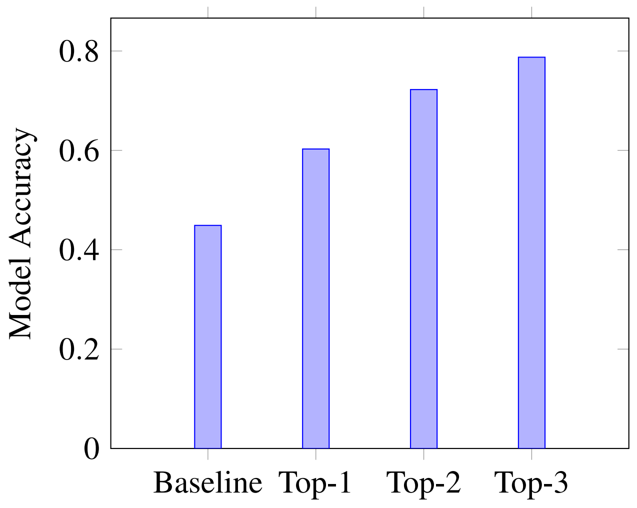


Figure 3.4: Fault localization accuracy. Baseline is the normal Python interpreter, and the Top-*k* bars represent our approach. The strong performance of our algorithm on a large, real-world dataset is the primary result of this chapter.

Users who are only willing to look at a single error message would have been better-served by our Top-1 model on this historical data. In addition, previous studies have shown that developers are willing to use very short ranked lists [PO11, Sec. 5.6], but not voluminous ones. Our Top-3 accuracy of 77% dramatically improves upon the current state of practice for scalable localization in Python.

3.3.4 RQ 2 — Feature Predictive Power

Having established the efficacy of our approach, we now investigate which elements of our algorithmic design (Section 3.2) contributed to that success. Table 3.1 summarizes the relative importance of the top features in our model. The features are ranked by their Gini importance (or mean decrease in impurity), a common measure for decision tree and random forest models [Bre01]. Informally, the Gini importance conveys a weighted count of the number of times a feature is used to split a node: a feature that is learned to guide more model classification decisions is more important. Similarly, Table 3.2 gives an alternate view of the relative feature importances, as measured by a standard analysis of variance (ANOVA). In both cases, we find

Table 3.1: Feature predictive power (for a Top-3 Decision Tree learned on the entire dataset). Parent and Child1–3 refer to the parent and first three children of the node in question.

Name	Category	Gini Importance
Parent size	Contextual (Syntactic)	0.112
Child3 is statement	Contextual (Syntactic)	0.061
Parent is list literal	Contextual (Syntactic)	0.055
Program crashes here	Dynamic (Error location)	0.039
Type is unknown	Dynamic (Type)	0.037
Parent type is unknown	Contextual (Type)	0.037
Err message is IndexError	Dynamic (Error message)	0.033
Size	Syntactic	0.028
Parent is dictionary	Contextual (Syntactic)	0.027
Child1 size	Contextual (Syntactic)	0.024
Is variable	Syntactic	0.020
Child1 type is unknown	Contextual (Type)	0.020

Table 3.2: Feature predictive power as measured by ANOVA. The F-score is shown in thousands to two significant figures; all p-values are less than 0.01. Parent and Child1–3 refer to the parent and first three children of the node in question.

Name	Category	F-score
Child3 is statement	Contextual (Syntactic)	69
Size	Syntactic	63
Program crashes here	Dynamic (Error location)	50
Parent type is list	Contextual (Type)	36
Child1 type is bool	Contextual (Type)	30
Parent type is unknown	Contextual (Type)	28
Child1 is expression	Contextual (Type)	25
Child3 size	Contextual (Syntactic)	20
Child1 is binary or unary op	Contextual (Syntactic)	20
Child1 size	Contextual (Syntactic)	18
Parent type is dictionary	Contextual (Type)	17
Type is string	Dynamic (Type)	12

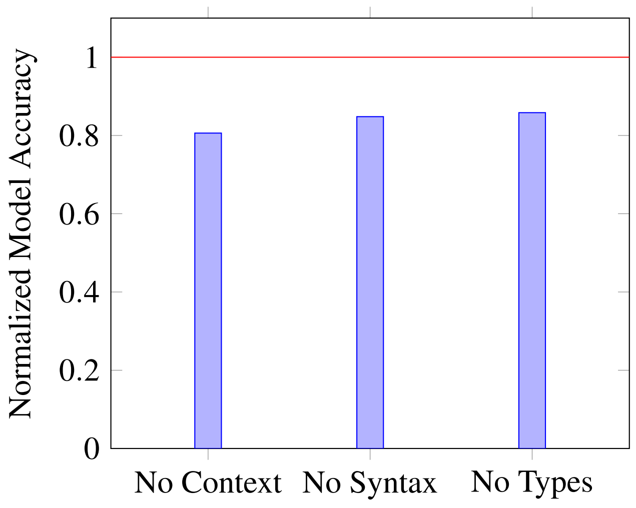


Figure 3.5: Normalized accuracy when categories of features are removed, based on Top-1 decision trees on a random subset of 20,000 program pairs, as compared to a Top-1 decision tree trained on all features.

that a mixed combination of static, dynamic, and contextual features are important: no single category alone suffices.

To support that observation, we also present the results of a leave-one-out analysis in which entire categories of features are removed and the model is trained and tested only on those that remain. Figure 3.5 shows that when the model is trained without typing, syntactic, or contextual features, the model's accuracy drops by 14%, 15%, and 19% respectively. In addition, we note that leave-one-out analyses underestimate importance in cases of feature overlap. For example, if a small program contains a string bug but only a few string variables, both type information and slice information may implicate similar terms.

We generally conclude that syntactic, dynamic, and contextual features (i.e., the design decisions of our algorithm) are crucial to our algorithm's accuracy.

Discussion of top features. Some of the specific features appearing in Table 3.1 and Table 3.2, such as "program crashes here" and features related to term size, are fairly direct. Others, such as "Child3 is statement", a contextual, syntactic feature that was found to be very important, merit additional explanation. In practice, we found it to encode complex control structures, like large loops, where fixing the crash often involved adding or removing whole statements from the loop. In contrast, features that determine if the parent node is a list or dictionary are relevant as strong negative features: we find that fixing bugs in beginner-written Python rarely requires changing the elements of a list or dictionary literal. Finally, we note the importance of our `Unknown` type feature for expressions that are never successfully evaluated in the given trace. The "Parent type is unknown" feature seems to help in the case where the bug is in an immediate child of the node that actually crashes. For example, if the programmer writes `x[i]` when it should have been `x[i-1]` and this causes an `IndexError` (array index out of bounds), then the bug is in the index `i`, and its parent's type is unknown because it is the node that crashed.

3.3.5 RQ 3 — Defect Categories

We investigate the sensitivity of our algorithm to different categories of Python errors: does PABLO apply to many kinds of novice defects? We investigate training and testing Top-1 decision trees on only those subsets of the dataset corresponding to each of the five most common uncaught exceptions: `TypeError`, `IndexError`, `AttributeError`, `ValueError`, and `KeyError`. Together, these five exceptions are the most common faced by our novices, making up 97% of the errors in our dataset (54%, 23%, 11%, 7%, and 3%, respectively).

As shown in Figure 3.6, these per-defect models have normalized accuracy between 86% and 115% of a comparable model trained on the dataset as a whole. This shows that our algorithm is robust and able to give high-accuracy fault localizations on a variety of defect types. Having consistent, rather than defect-type-sensitive, performance is important for debugging-tool usability [PO11, BBC$^+$10, AP10].

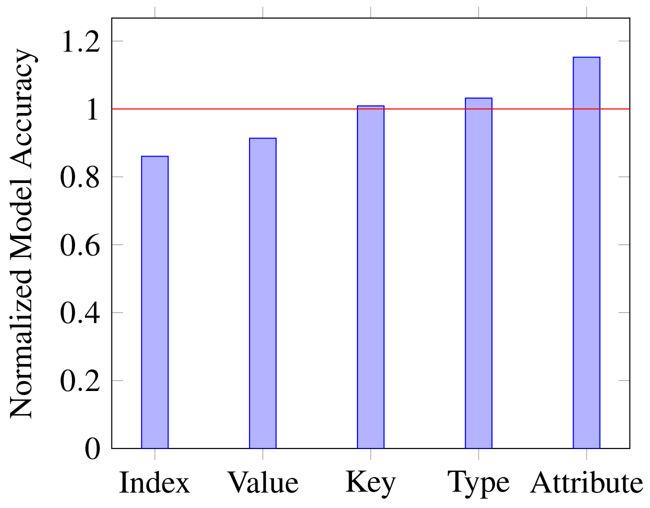


Figure 3.6: Accuracy when a Top-1 decision tree is trained and tested on only the data exhibiting the error on the x-axis, as compared to a Top-1 decision tree on all the data.

3.3.6 RQ 4 — Diversity of Programs

To demonstrate that our evaluation dataset is not only larger but also more diverse than those used in previous work, we compare the diversity of programs used here to those in a relevant baseline. The NATE algorithm [SSC+17] also provides error localization using a machine-learning approach, and its evaluation also focused on beginner-written programs. However, NATE targets strongly statically typed OCaml programs: submissions to just 23 different university homework problems. Such a dataset is comparatively homogeneous, raising concerns about whether associated evaluation results would generalize to more diverse settings. In contrast, in our PythonTutor dataset, users were not constrained to specific university assignments. We hypothesize that the data are thus more heterogeneous. To assess this quantitatively, we used agglomerative clustering to discover the number of "natural" program categories present in both our dataset and the NATE dataset. Datasets with more natural program categories are more heterogeneous.

Distance Metric

Many clustering algorithms depend on distance metrics. To measure the distance between programs, we flattened their ASTs into strings of tokens, and then computed the Levenshtein edit distance [Lev66]. We do not compute an AST distance directly since that is less tractable on our large dataset (i.e., cubic [PA16] instead of Levenshtein's quadratic [WF74]). Levenshtein distance is not a good *absolute* measure of program diversity since similar programs can have different tree structures, but it does show *comparative* diversity.

The flattening process, which is based on standard approaches for tree-structured data [Cau], is defined recursively: the string for an AST begins with a unique token corresponding to the AST node type, which is followed by the concatenation of the transformations of the tree's subtrees. The string ends with an "end" token to denote the end of the tree. This embeds information related to the tree structure of the AST in the flattened string.

Clustering Algorithm

We performed agglomerative clustering on the datasets of flattened programs [Mül11]: every datapoint starts in its own cluster, and the two closest clusters are merged until there are no clusters that differ by less than some threshold. We used a single linkage approach in which the distance between clusters is the minimum distance between their elements. To account for differences between Python and OCaml, we z-score nodes against not only others at the same tree depth, but also against others one or two levels below [Zah71]. This makes cluster counts at each threshold value comparable. Our implementation uses the standard SCIPY library (**`scipy.cluster.hierarchy.fcluster`** with the **`inconsistent`** method).

Figure 3.7 compares the NATE dataset to a random sample of equal size from our dataset. At a high level, the structural similarity in the graphs confirms the comparability of cluster counts between the two datasets. For all values of the inconsistency threshold, there are at least 48% more clusters in our sample than in the OCaml dataset. This suggests that our dataset contains a

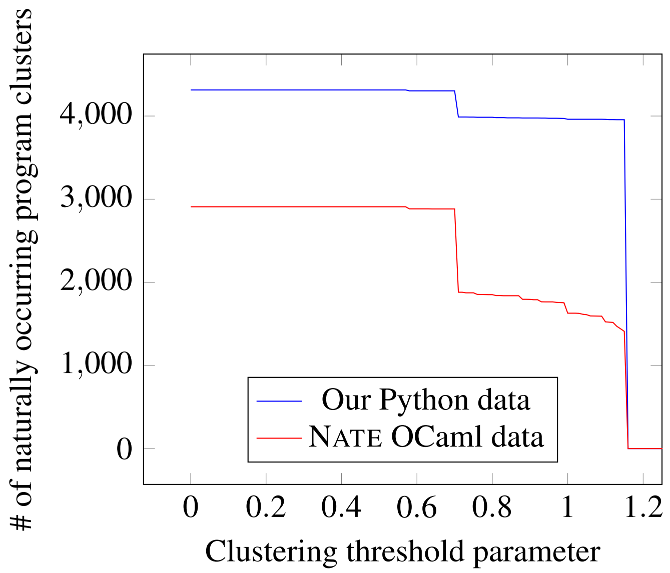
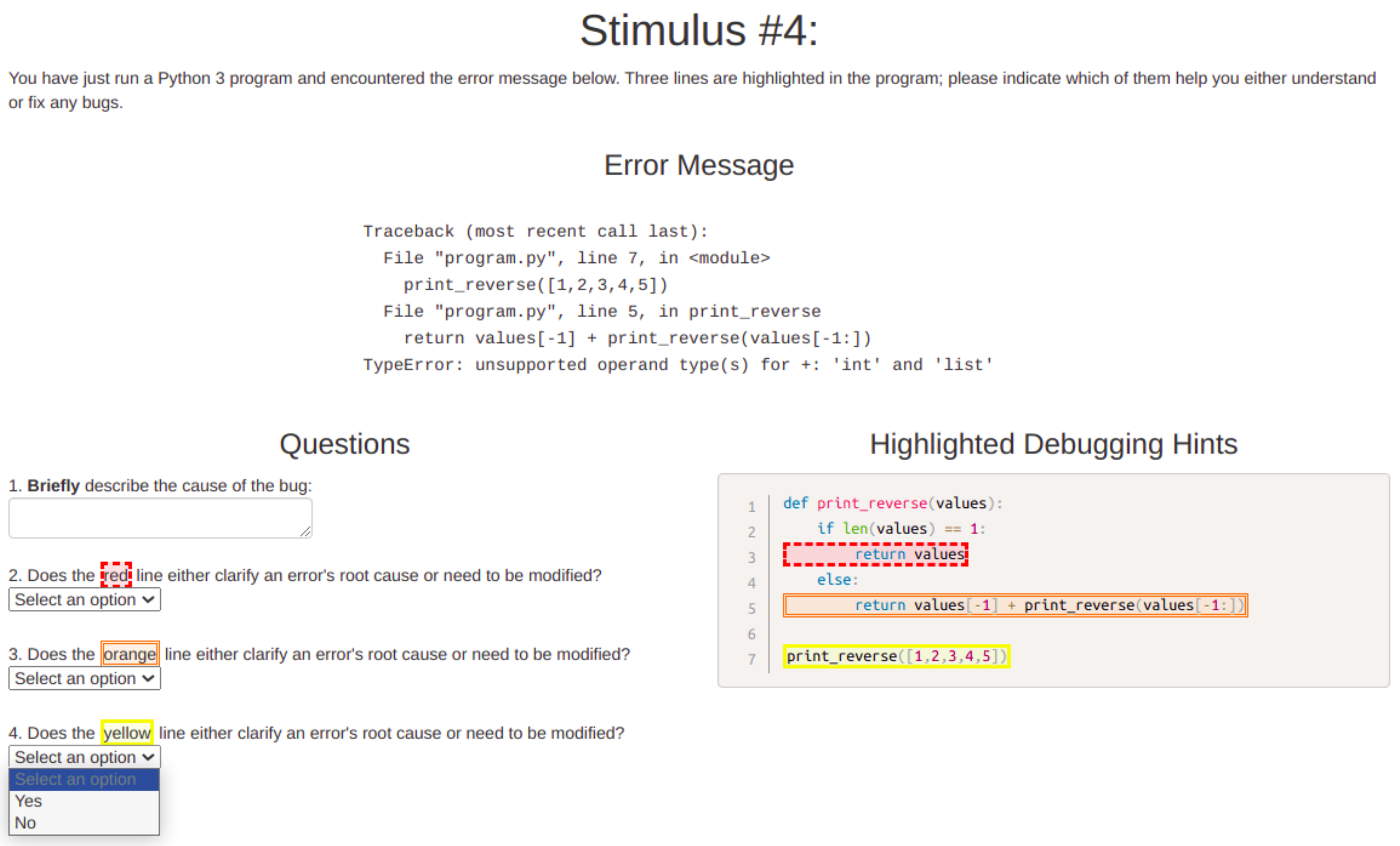


Figure 3.7: Number of natural program clusters in the NATE dataset and in a random sample of our dataset of the same size, as a function of clustering parameter (informally, how "strict" or "tight" the clustering is). At all points our dataset demonstrates greater diversity.

more diverse set of programs. We are not claiming any advances in clustering accuracy in this determination (indeed, scalability concerns limited us to coarser approaches); instead, our claim is that even with simple clustering, it is clear that our dataset contains a greater diversity of programs, even when controlling for size, than were considered by previously-published evaluations. We view it as an advantage of our algorithm that it can apply to many different program categories.

3.3.7 RQ 5 — Multi-Edit Bug Fixes

In addition to the automated metrics described above, we also evaluate PABLO in an IRB-approved human study. We selected 30 programs at random from the PythonTutor dataset, and presented each with 3 highlighted lines representing the Top-3 output of PABLO (see Figure 3.8). For this study we worked at the granularity of lines rather than of expressions to simplify the presentation of three distinct and non-overlapping localizations for comparison. Each participant was shown a random 10 of these annotated-program stimuli and asked, for each highlighted line, whether it "either clarifies an error's root cause or needs to be modified?" Not all participants



Stimulus #4:

You have just run a Python 3 program and encountered the error message below. Three lines are highlighted in the program; please indicate which of them help you either understand or fix any bugs.

Error Message

```
Traceback (most recent call last):
  File "program.py", line 7, in <module>
    print_reverse([1,2,3,4,5])
  File "program.py", line 5, in print_reverse
    return values[-1] + print_reverse(values[-1:])
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Questions

1. **Briefly** describe the cause of the bug:

2. Does the red line either clarify an error's root cause or need to be modified?
Select an option

3. Does the orange line either clarify an error's root cause or need to be modified?
Select an option

4. Does the yellow line either clarify an error's root cause or need to be modified?
Select an option
Select an option
Yes
No

Highlighted Debugging Hints

```
1  def print_reverse(values):
2      if len(values) == 1:
3          return values
4      else:
5          return values[-1] + print_reverse(values[-1:])
6
7  print_reverse([1,2,3,4,5])
```

Figure 3.8: An example stimulus from the user study.

```
1 def devowel(word):
2     w_list = list(word)
3     vowels = ['a', 'e', 'i', 'o', 'u']
4     for letter in w_list:
5         if letter in vowels:
6             w_list = w_list.remove(letter)
7     return ''.join(w_list)
8 print(devowel('foobar'))
```

Figure 3.9: A program with a bug on line 6: `remove` edits in place and returns `None`

answered all questions, but we were able to use data from 42 participants in our analyses.

Overall, participants find the first and second lines from PABLO useful 75% and 28% of the time; at least one of PABLO's top three is useful 84% of the time. On the other hand, participants find the line indicated by Python's error message helpful only 77.3% of the time. That is, the output of PABLO outperforms vanilla Python by 6.5% ($p = 0.018$, two-tailed Mann-Whitney test).

When considering only the 14 programs with complex bugs where the original novice programmer made edits to multiple lines, humans find PABLO even more helpful; PABLO's first and second lines are helpful 79% and 36% of the time, and at least one of the top three is useful 89% of the time. We observe that multi-edit bugs are quite common, accounting for almost half the bugs in our data set. For these complex multi-edit bugs, the Python interpreter alone provides novices limited support while PABLO provides additional useful information more than one-third of the time.

3.3.8 Qualitative Analysis

We now highlight a few indicative localization examples in detail to demonstrate how our algorithm accurately localizes faults. These examples are simplified slightly for presentation and to protect the anonymity of the programmers, but they retain the essential character of the originals from our dataset.

```
1 areaCodes = [800, 555]
2 for i in range(0, len(areaCodes) + 1):
3     print(areaCodes[i])
```

Figure 3.10: A program with an off-by-one bug on line 2.

NoneType. The function in Figure 3.9 attempts to remove all vowels from a given word. However, the assignment on line 6 actually replaces the entire word with `None`, because the `remove` method modifies the list in place and does not have a return value (i.e. returns `None`). The user-corrected version forgoes the assignment and just calls `w_list.remove(letter)`. In the buggy case, Python does not crash until line 7, where its message is the somewhat-misleading `TypeError: can only join an iterable`. However, PABLO flags the correct statement, based on the features that it is an assignment statement whose second child has type `NoneType`. This captures the intuition that there is rarely a good reason to assign the value `None` to a variable in novice programs.

Note that PABLO uses both the syntactic form of the statement (an assignment) and the dynamic type of one of its children, so all our feature categories — syntactic, dynamic, and contextual — were useful here.

Off-by-one bugs. In Figure 3.10, the programmer incorrectly adds one to the high end of a range, causing an `IndexError: list index out of range` on line 3 during the final iteration of the for-loop. The correct expression to blame is the addition `len(areaCodes) + 1`. Some of the features PABLO uses to correctly localize this bug include that the error is an index error, the type of the parent, and the fact that it is an addition expression. This example also highlights the use of all three categories of features simultaneously to capture a notion of root cause that more closely aligns with human expectations.

3.3.9 Threats to validity

Although our evaluation demonstrates that our algorithm scales to accurately localize Python errors in large datasets of novice programs, our results may not generalize to all use cases.

Overfitting. Since our algorithm makes use of supervised machine learning, one threat to validity is overfitting (i.e., learning a model that is too complex with respect to the data and thus fails to generalize). We mitigate this threat by using random forest models, as discussed in Section 3.3.2.

Language choice. We have only demonstrated that our technique works for Python 3. We hypothesize that it should apply to similar dynamically-typed languages, such as Ruby, but such evaluations remain future work. We mitigate this threat slightly by focusing on a subset of Python which does not include relatively "exotic" features such as generators, **`yield`**, and coroutines that may not always be present in other languages.

Target population. Unlike many classroom studies of students, we have less information about the makeup of our subject population. The general popularity of PythonTutor is an advantage for collecting a large, indicative dataset, but it does mean that we have no specific information about the programmers or the programs they were trying to write. In general, while the website is used by many classes, most of the users appear to be non-traditional students; our results may apply most directly to that population.

Ground truth. The size of our dataset precluded the manual annotation of each buggy program. Instead, we used historical successful edits from actual users as our ground truth notion of the desired fault localization. This has the advantage of aligning our algorithm with user intuitions in cases where there are multiple logically-consistent answers (see Section 3.1), and thus increasing the utility of our tool. However, this definition of ground truth may be overly permissive: the next correct program in the historical sequence may not contain a true bug fix at all (as an extreme example, the user may have just deleted the whole program, which would count as a fix since the blank program does not crash), or it may contain both a true bug fix and also additional spurious changes beyond those strictly needed to fix the bug. We mitigate this threat by discarding as outliers program pairs that had very large relative changes.

3.3.10 Evaluation Summary

PABLO is *helpful*, providing high-accuracy fault localization that implicates the correct terms 59–77% of the time (for Top-1 to Top-3 lists, compared to the baseline Python interpreter's 45% accuracy) and outperforming the baseline ($p = 0.018$). PABLO is *general*, performing similarly on the top five exceptions that make up 97% of novices crashes, and providing additional helpful information for multi-line fixes 36% of the time (compared to the baseline Python interpreter's 0%).

In addition, we investigated our algorithm's design decisions, finding that all our categories (i.e., static, dynamic, and contextual) were critical, and we analyzed the dataset to determine that it was more heterogeneous than that of related work. Our evaluations involved over 980,000 pairs of beginner-written Python programs as well as a direct human study of 42 participants.

3.4 Related Work

Broadly, the two most relevant areas of related work are software engineering approaches to fault localization (typically based on dynamic test information) and programming languages approaches to fault localization (typically based on static type information). Fault localization has only increased in relevance with the rise of automated program repair [Mon18, LNFW12, NQRC13], where many techniques depend critically on accurate fault localization [QMLW13].

A significant body of work in fault localization follows from the Tarantula project [JH05]. Jones *et al.* proposed that statements executed often in failing test runs but rarely in successful test runs were likely to be implicated in the defect. Such spectrum-based approaches gather coverage or other dynamic information and rank statements by a mathematically-computed suspiciousness score. Subsequent algorithms such as Pinpoint [CKF$^+$02] or Ochiai [AZG06] have improved upon Tarantula in various ways. For example, the Cooperative Bug Isolation approach gathers richer dynamic information (i.e., invariants) and also combines information from multiple end-user sources [LNZ$^+$05]. Spectrum-based fault localization has been the subject of significant empirical evaluation [AZGvG09]. Research on spectrum-based methods continues to this day, with refinements, and even optimality results, to the associated mathematical formulae [YXK$^+$17]. Projects like Multric [XM14] have used machine learning to combine these spectrum-based suspiciousness scores based on empirical data, and Savant [BLLLGG16], TraPT [LZ17], and FLUCCS [SY17] refine this process by also using, respectively, inferred invariants, mutation testing, and code metrics like age and churn. CrashLocator [WZCK14] computes suspiciousness scores without needing positive test cases, but it requires a large suite of crashing cases as well as an oracle that groups crashes by similarity.

PABLO shares the overall goal of spectrum-based fault localization: pinpointing suspicious statements to reduce debugging effort. However, where spectrum-based approaches traditionally focus on industrial-scale programs, we target beginner-written software. Spectrum methods require multiple test cases (ideally very many of them); we use just one program execution and our work is aimed at novices who may not even be familiar with the notion of test suites. Spectrum methods focus heavily on dynamic features; we make critical use of syntactic, contextual and type information as well. Indeed, the machine learning based approaches above use features that are entirely disjoint from ours and inapplicable in our setting, with the sole exception of the code complexity metrics of FLUCCS. Unlike spectrum features, many of our features have no obvious connection to faultiness, so our surprising positive result is that we can still use these features to localize faults in our domain. In addition, some human studies have focused specifically on accuracy and expertise for fault localization [FW10, PCR$^+$03, RBR05]; our decision to use features to support novices is informed by such insights.

Zeller's popular Delta Debugging algorithm [Zel99], interpreted generally, efficiently finds a minimal "interesting" subset from among a large set of elements. When the set of elements represents changes made to a source code version control system and interesting is defined with respect to failing a test suite, it can quickly locate program edits that cause regressions. Alternatively, when considering correct and failing execution states, such approaches can help focus on variable values that cause failures [CZ05]. While our dynamic slicing information can be viewed as a coarse approximation to the precise, fine-grained localization such an approach can provide, a key difference is our use of machine learning to agree with human judgments in cases where multiple causes are equally logically valid.

In the programming languages community, a large body of work has focused on localizing faults and providing better error messages, typically through the use of type information. In [PKW14b], they localize multiple type errors by using a MaxSMT solver to find a minimal set of typing constraints that must remain unsatisfied. By adding weights to the constraints, they can customize what "minimal" means, allowing some constraints to have higher weight or even be "hard" constraints with infinite weight. For example, constraints corresponding to library function declarations should be hard because the bug is presumed not to be in the current code rather than the library definitions. Similarly, Seminal [LGC06] uses enumerative search to find a minimal set of expressions with incompatible types, and SHErrLoc [ZM14] identifies a set of incompatible type constraints, then chooses the likely type error using the Bayesian heuristic that the program is probably mostly correct. In PABLO we rely on the complementary heuristic that faults will share characteristics with previously recorded faults, and apply this heuristic to features beyond just type information. Mycroft [LCSS16] modifies existing type inference algorithms to produce a "correcting set" for a program with a type error. Mycroft assumes that the minimal such set is the most desirable to the user, and has no way to rank multiple equally small sets. We instead use machine learning to agree with human judgements.

PABLO is most directly inspired by the NATE [SSC$^+$17] system, which introduced the notion of training classifiers over pairs of buggy-and-fixed programs, and uses machine learning on static and contextual features to localize type errors in OCaml code. They generate a similar dataset to ours by instrumenting an IDE to take snapshots of students' code over time as the students solve programming assignments. Then whenever the code does not compile due to a type error, they treat the next version which *does* compile as the fixed version of that error.

However, this work was limited to purely functional OCaml programs where the static type discipline was crucial in both restricting the class of errors, and providing the features that enabled learning. In PABLO we employ a similar approach to localize Python faults, but we use dynamic features as well as static and contextual ones, we handle a variety of errors beyond type errors, and we evaluate our approach on a set of programs far more heterogeneous and more than two orders of magnitude larger.

3.5 Conclusion

We present an approach for accurately localizing novice errors in off-the-shelf, beginner-written Python programs. Our approach uses a combination of static, dynamic and contextual features. Static features, such as syntactic forms and expression sizes, are a particularly powerful heuristic for novice programmer defects. Dynamic features can both implicate relevant terms and rule out irrelevant program regions. Contextual features allow our approach to gain the benefits of precise AST- or CFG-style information while retaining scalable performance. We use off-the-shelf machine learning to accurately combine those disparate features in a way that captures and models human judgments of ground-truth correct answers — a notion that is especially relevant when multiple program locations are equally formally implicated but not equally useful to the user.

We desire an approach that is *helpful*, *general* and *scalable*. We evaluate our approach with respect to historical defects and fixes. All feature categories (static, dynamic, and contextual) were relevant to success, as measured by multiple analyses (Gini, ANOVA and leave-one-out).

Our evaluation demonstrates significant scalability and generality. Our 980,000 instances were two orders of magnitude more numerous than similar related work and measurably more diverse; we augmented our dataset with a direct human study of 42 participants. Ultimately, PABLO was quite accurate, implicating the correct program location 59–77% of the time (compared to the Python interpreter's 45% accuracy), outperforming the baseline ($p = 0.018$) and providing additional useful information 36% of the time (compared to Python's 0%).

Acknowledgements for Chapter 3

This chapter, in full, is adapted from "PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization" in the proceedings of the 2020 Technical Symposium on Computer Science Education (SIGCSE), by Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer [CES$^+$20], as well as from an earlier version that was submitted for publication to ESEC/FSE 2019. The dissertation author was the primary investigator and author of this material.

Chapter 4

Pyrite: Analytic Program Repair for Python Novices

4.1 Introduction

In the previous chapter, we showed how to localize errors in novice Python programs. In this chapter, we focus on *repairing* errors in novice Python programs instead of just localizing those errors, in order to provide richer debugging hints than can be generated from error locations alone. As in the previous chapter, there are two steps - finding repairs and then choosing how to turn those repairs into appropriate debugging hints for novices - and we leave the second step for future work.

We seek to repair novice programs in the setting where we do not have a testing suite or indeed know at all what program the novice was trying to write. Thus techniques which require a known program specification - whether to use other students' bug fixes on that same problem [HGS$^+$17b], scale expert feedback by providing the same feedback to similar submissions [HGS$^+$17b], or compare submissions to known-good solutions [SGSL13b, GRZ18, WSS18] - are not applicable here.

Instead, we extend the key insight from the previous chapter by hypothesizing that similar errors will frequently have similar fixes, even if the errors occur in the context of very different programs. We thus propose to repair programs by learning repairs from a dataset containing examples of students fixing their own bugs. Proceeding similarly to the previous chapter and to prior work [SEC+20], our system PYRITE:

1. takes as input a set of pairs of programs, each representing a program that crashes and the fixed version of that program

2. computes a *bag of abstracted terms* (BOAT) [SSC+17] representation for each crashing program. For each term that differs between the buggy and fixed programs, the label now represents not only *that* it needs to be fixed but also the syntactic form of the fix (Section 4.2.1).

3. trains two classifiers on these vectors - one to predict whether a term is changed, and one to predict the syntactic form of that change

Then, given a new crashing program, PYRITE will

1. compute BOAT vectors for each AST node of the program

2. classify each vector to obtain the likelihood that each corresponding node is to blame for the crash, and what syntactic forms are most likely to be correct fixes

3. use these predictions to synthesize potential program repairs, ranked by likelihood

We evaluate PYRITE on the the dataset used by PABLO (with a few changes discussed in Section 4.3) - more than 980,000 programs from PythonTutor [Guo13]. We find that PYRITE can generate repairs for 76% of these novice programs. Additionally, we evaluate the helpfulness of our generated repairs in an IRB-approved user study with 41 participants. We find that in almost half of the feedback PYRITE provides, users indicate that the feedback contains additional useful information beyond that given by just the default error message of the Python interpreter.

4.2 Algorithm Overview

We present PYRITE, an algorithm for repairing novice Python programs that crash at runtime. This algorithm extends PABLO by using machine learning models based on static, dynamic, and contextual features to predict both which program terms need to be changed as well as what the syntactic form of the changes should be (Section 4.2.1 and Section 4.2.2). We then use an enumerative synthesis algorithm to create concrete repairs which are presented to the programmer (Section 4.2.3).

4.2.1 Fix Templates

As in the previous chapter, we compute BOAT vectors for each term in the buggy program, as well as a label representing whether the term is the error location. However in order to guide repair synthesis, we now compute an additional label for buggy terms: the *fix template*. We desire this label to be specific enough to guide synthesis, yet general enough that we can learn patterns without overfitting. We expect that syntactic patterns will generalize well, but the exact edits will not. For example, if a student fixes a program by replacing a misspelled variable with its correct spelling, that exact text diff may be unique in the dataset, but the pattern of replacing an (undefined) variable with another variable will likely recur.

Following prior work [SEC$^+$20], we thus use Generalized Abstract Syntax Trees (GASTs) as our labels. The GAST for a given program term is like the normal AST representing that term, except that it is *generalized* in the following ways:

- All concrete variables *a,b,c, etc* are abstracted to a single generalized variable $\hat{x}$. Similarly, each of the following categories is abstracted to a single generalized member: literals, unary operators, binary operators, field names, module names, and function names.
- For tuples, lists, and all other nodes that admit an arbitrary number of children, we remove all such children after their first.

For example, the expression `(a + b) - 1` would be represented by the template $(\hat{x} \oplus \hat{x}) \oplus \hat{n}$. Additionally, to represent the pattern of reusing the old expression in the fixed one, we add one more template *orig* - for example, if the user had changed `(a + b)` to `(a + b) - 1`, then instead of using the template above we would use $orig \oplus \hat{n}$. If a term is deleted entirely, the repair template is the empty GAST.

4.2.2 Machine learning

We now train two machine learning models. The first corresponds to the localization model from PABLO, while the second uses the same features to try to predict a conditional probability: for a given location, *given* that it should be changed, what syntactic form (GAST) will the change take according to our historical dataset? As in Chapter 3, we assume the existence of a dataset of program pairs, where each pair consists of a buggy program (one that exhibits a runtime crash) and a fixed version of that same program. (See Section 4.3 for how we construct such a dataset.)

Features and labels. We compute a BOAT vector of syntactic, dynamic, and contextual features for every statement and every expression in each buggy program, using the same features as PABLO (Section 3.2.2), and we compute a tree-diff [LLL09] between the buggy and fixed programs to label which terms changed between the two versions. However, for PYRITE we change the labels in two ways.

Firstly, whenever a term and each of its subterms appear in the diff, we only consider the larger term to have been changed, not the subterms. For PABLO's error localization, we considered implicating a changed subterm as still a success. This was because we assumed that in the domain of localization, pointing to any part of a buggy term is enough to guide the user towards fixing that term (and overall such design choices were validated by the user study in Section 3.3 in which users found our localizations helpful). However for PYRITE, we seek to repair the program. Assuming the historical user's entire edit was necessary to repair the program,

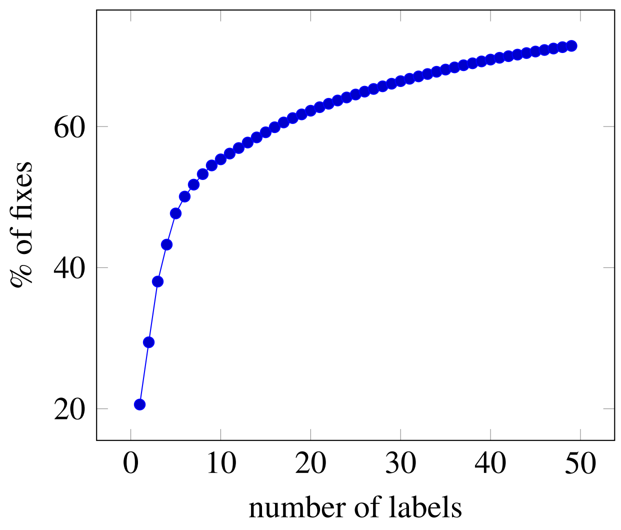


Figure 4.1: Cumulative number of fixes described by the first *n* fix template labels.

then implicating a sub-expression of the changed term is not good enough since no repair to that sub-term will repair the program as a whole.

Secondly, in addition to the boolean label from Chapter 3 indicating whether the term was changed or not, we add a new label which has two parts:

1. The syntactic form of the new term, represented by a GAST (Section 4.2.1).
2. A boolean flag indicating whether the repair *replaces* the given term, or is instead an *insertion* next to that term.

Predicting this new label is a multi-class classification problem. To prevent the number of classes from being too large, we retain only the 49 most common labels observed in our dataset (which account for 71.4% of the total - see Figure 4.1), and replace all labels rarer than those with a special 50th label `Other`.

4.2.3 Repair Synthesis

In the previous sections, we discussed how we trained machine learning models on our dataset to predict error locations and fix templates. We now describe how we use those models to

Table 4.1: The most common fix templates observed in the dataset.

Template	Prevalence (%)
replace by variable	20.6
insert variable	8.8
replace by literal	8.6
insert literal	5.2
delete the term	4.4
insert assignment of literal to variable	2.4
insert function call on one variable	1.7

guide repair synthesis for a new buggy program.

Given a new buggy program, we first extract BOAT vectors for every expression and statement in the program, as in the training phase. However, these vectors do not have labels - either error locations or fix templates - since we do not have a fixed version of the program to derive those from. Instead, we run our two machine learning models on the unlabeled vectors to get, for each vector, a confidence score representing how likely that program location is implicated in the bug, as well as a list of predicted fix templates representing the most likely ways that location could be repaired.

Synthesis proceeds as follows. First, we list the 5 outputs of the localization model with the highest confidence. For each of those potential fix locations, we list the top 6 outputs of the fix template model. Now for each of these 30 location/template pairs, we enumerate program terms that match the template. Note that there are frequently impractically many or even infinitely many possible terms matching a single fix template. In particular, the template representing a single abstract variable could be instantiated with any concrete identifier (both those defined in the code and those defined by Python itself), and the template for a single literal could be instantiated with any concrete literal of the appropriate type. For variables and literals, we thus construct a finite list of terms to try by gathering options from two sources: we try all variables or literals that were used in the buggy input program itself, and also the 20 most common variables/literals that appear in all fixes in our training set. (For the list of variables commonly used in fixes, see Table 4.2.)

Table 4.2: The most common variables appearing in historical fixes from the training set. Frequency is measured relative to the most common (len), so e.g. the variable x appears in around a quarter as many historical fixes as len does. All of these are Python built-in function names, except for the common user-defined variable names i and x.

Identifier	Relative Frequency
len	1
range	.91
str	.73
int	.73
print	.68
append	.62
list	.34
i	.29
x	.26
input	.26

We filter the potential patches enumerated above to actual repairs by using each one to patch the program and then checking if the program still crashes when run (with either the old exception or any new one). Additionally, repairs are rejected if the resulting program does not call all user-defined functions, under the heuristic that with short, novice-written programs, any function the user wrote is one they intended to call, and so a repair that does not call them is likely a false fix (e.g. deletes buggy code or redirects control flow around it, rather than fixing it - see Section 4.3).

All remaining repairs do cause the program to run without crashing, but some may better match programmer intent than others, and thus may be more useful as debugging hints. To select the repairs most likely to align with human judgement, we thus sort the set by the size of the tree-diff [LLL09] between the buggy version and our patched version, returning the smallest edits first. This corresponds to the heuristic that the user's existing code is mostly correct and matches their intent.

4.3 Evaluation

We conducted both a large-scale empirical evaluation of PYRITE and also a human study to address these research questions:

RQ1 Do we successfully synthesize repairs for bugs in novice Python programs?

RQ2 Do humans find our algorithm useful?

To answer the first question, we conducted a large-scale empirical evaluation of our tool on the PythonTutor dataset, and to answer the second question, we conducted an IRB-approved user study using sample of 30 programs from that dataset. Our raw data consist of the same programs from Section 3.3, with two changes:

- We no longer exclude programs with undefined variables (Python's NameError or UnboundLocalError). Excluding those errors turned out to be an unnecessary design decision in PABLO.
- We now exclude program pairs where the fixed version does not call all user-defined functions at runtime. We do this because empirically, such pairs very frequently correspond to a "fix" that clearly does not match the programmer's intent. For example, the original code may at the top level just call a buggy function in a loop; changing the loop condition such that the loop no longer runs even once and the function is never called will prevent the program from crashing, but it does not truly repair the program.

4.3.1 RQ 1 — Success rate

We ran PYRITE on buggy programs from the testing data. As shown in Figure 4.2, overall we were able to synthesize repairs for 76.1% of these programs; for the rest, the synthesizer either terminated with no output or timed out (timeout set to 1 minute). In 9.4% of the programs where

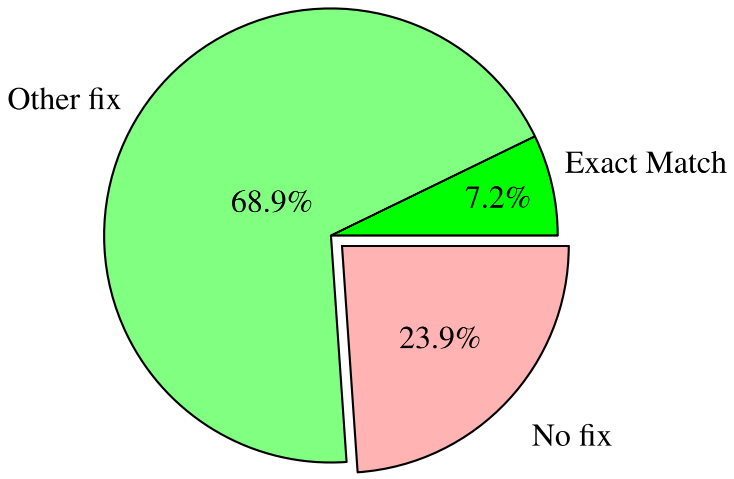


Figure 4.2: We successfully synthesize repairs for 76% of input programs, occasionally producing the exact repair the historical student chose.

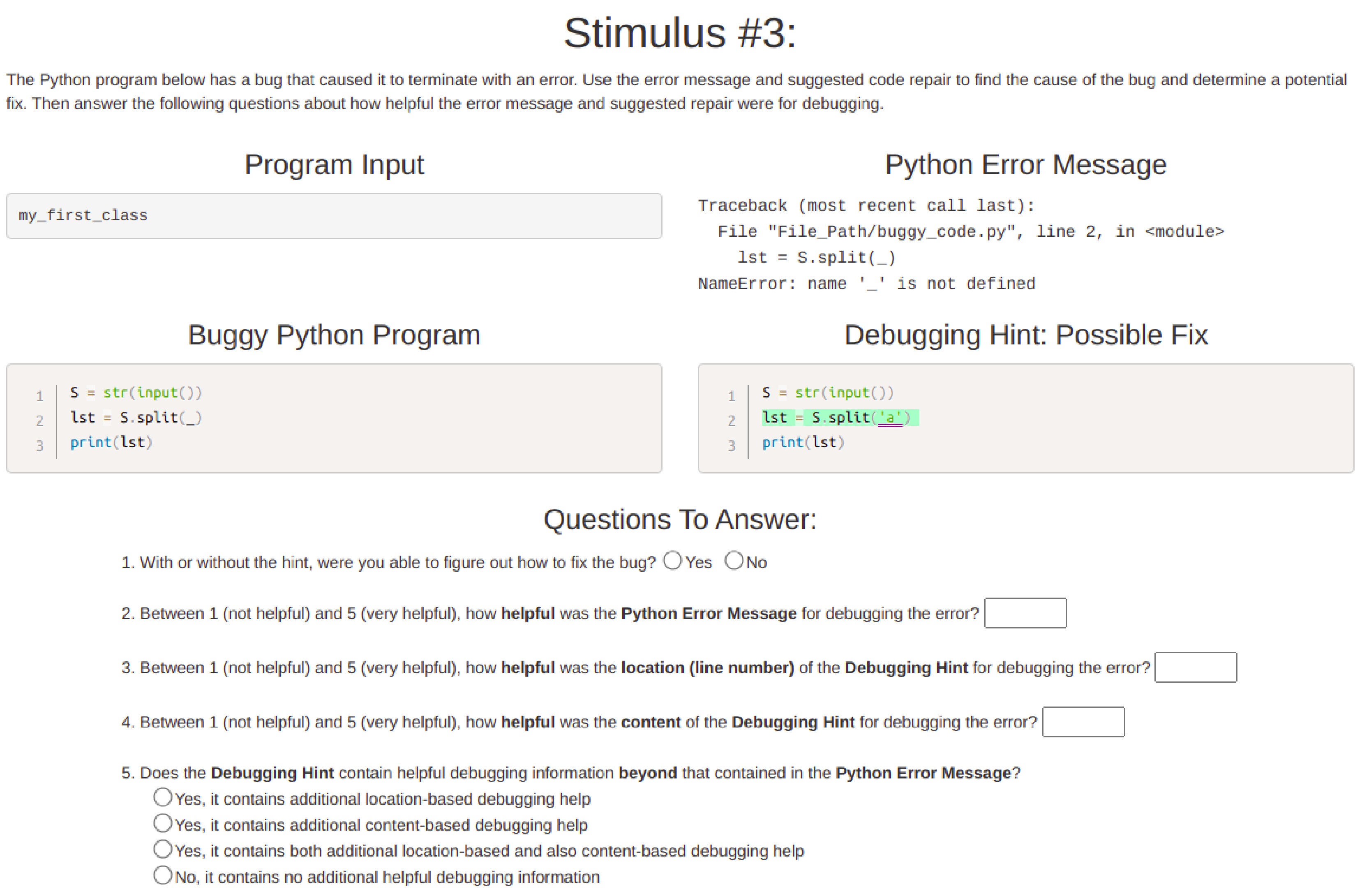
we were able to synthesize a repair, our repair was exactly the same as the one from the historical dataset. We thus establish that in the vast majority of cases, we can successfully synthesize repairs for novice Python programs.

4.3.2 RQ 2 — Helpfulness

Having established that we can usually find *some* repair, the next question is whether those repairs are actually *helpful* to novices. To evaluate how helpful our synthesized debugging hints are, we conducted an IRB-approved user study. We collected usable data from 41 participants, mostly evenly divided between computer science undergraduates, graduate students, and professionals.[1] All had at least 2 years of general programming experience, with a median of 6 years, and all but one had at least one semester of Python experience.

Each participant was shown a sequence of stimuli like the one in Figure 4.3. On the left is the buggy program from our dataset, along with the input it was run on (if any). On the right is the error message that the vanilla Python interpreter produces, along with a "debugging hint"

[1]The exact breakdown: 12 CS undergraduates, 2 other undergraduates, 13 CS graduate students, 12 professional software developers, and 2 professional CS researchers or teachers.



Stimulus #3:

The Python program below has a bug that caused it to terminate with an error. Use the error message and suggested code repair to find the cause of the bug and determine a potential fix. Then answer the following questions about how helpful the error message and suggested repair were for debugging.

Program Input

```
my_first_class
```

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    lst = S.split(_)
NameError: name '_' is not defined
```

Buggy Python Program

```
S = str(input())
lst = S.split(_)
print(lst)
```

Debugging Hint: Possible Fix

```
S = str(input())
lst = S.split('a')
print(lst)
```

Questions To Answer:

1. With or without the hint, were you able to figure out how to fix the bug? ◯ Yes ◯ No

2. Between 1 (not helpful) and 5 (very helpful), how **helpful** was the **Python Error Message** for debugging the error?

3. Between 1 (not helpful) and 5 (very helpful), how **helpful** was the **location (line number)** of the **Debugging Hint** for debugging the error?

4. Between 1 (not helpful) and 5 (very helpful), how **helpful** was the **content** of the **Debugging Hint** for debugging the error?

5. Does the **Debugging Hint** contain helpful debugging information **beyond** that contained in the **Python Error Message**?
   - ◯ Yes, it contains additional location-based debugging help
   - ◯ Yes, it contains additional content-based debugging help
   - ◯ Yes, it contains both additional location-based and also content-based debugging help
   - ◯ No, it contains no additional helpful debugging information

Figure 4.3: An example stimulus from the user study. This example shows a reasonably good repair generated by PYRITE; other participants were shown the historical novice's repair instead, which was the same except it used '_' instead of 'a'.

Table 4.3: Answer breakdown for stimulus question 5, representing 328 total responses for PYRITE stimuli and 287 for historical user fixes.

Answer choice	PYRITE (%)	Historical fix (%)
location	8	3
content	20	43
both	17	42
neither	55	13

that is either a) the historical student's fixed version of that program from our dataset, or b) the output of PYRITE. For each stimulus, the user was asked about the helpfulness of the Python error message and the debugging hint.

Participants' answers to stimulus question 5 are summarized in Table 4.3. For comparison, we show the answers for both PYRITE stimuli and for the actual historical fixes, but we note that in a real deployment of a debugging assistance tool like PYRITE those historical fixes would not be available, so those numbers represent a high bar rather than an obtainable baseline. Overall, users found that PYRITE's debugging hints provided additional useful information beyond that contained in the vanilla Python interpreter's error message 45% of the time (that is, any "yes" answer to Q5). Additionally, in the vast majority of these cases (37% of the total), the hint was helpful beyond merely providing a useful location, thus validating our decision to focus on providing repairs instead of just PABLO's error localization.

4.3.3 Qualitative Analysis

We conclude the evaluation by looking at two example stimuli from the human study.

Of the 30 programs examined in the study, Figure 4.4 shows the case where PYRITE did the worst, completely failing to produce a useful hint. The historical student's fix correctly resolves the bug by removing the incorrect uses of "`self.`". PYRITE, however, resolves the bug by inserting a new line that sets `counter` to the empty string, thus preventing the buggy loop from running at all. Thus while study participants rated the human repair to be quite helpful, 100%

```python
counter = ['this', 'is', 'spot', 'see', 'spot', 'run']
count = {}
for item in counter:
  self.count[item] = self.count.get(item, 0) + 1
```

(a) Original (buggy) program

```python
counter = ['this', 'is', 'spot', 'see', 'spot', 'run']
count = {}
counter = ''
for item in counter:
  self.count[item] = self.count.get(item, 0) + 1
```

(b) PYRITE suggested fix

```python
counter = ['this', 'is', 'spot', 'see', 'spot', 'run']
count = {}
for item in counter:
  count[item] = count.get(item, 0) + 1
```

(c) Historical student fix

Figure 4.4: An example from the user study where PYRITE performs poorly.

of study participants agreed that PYRITE's repair was completely unhelpful (score 1 out of 5 on questions 3 and 4).

Figure 4.5 shows a case where PYRITE does well. The buggy program on the left crashes on line 2 because `c` is not defined in the math library. As `c` is defined by the programmer anyway on line 9, PYRITE correctly resolves this bug by deleting the faulty import statement. Interestingly, the historical student's fix is less good: importing `pi` instead of `c` makes the program run, but since `pi` is never used, it is better style to instead import nothing at all. Overall, user study participants rated PYRITE's fix here as better than the student's one (an average answer on question 3 of 4.3 vs 2.4), though this difference was not statistically significant when controlling for multiple comparisons.

```python
from operator import (add, mul)
from math import c
a = add
b = mul
print(mul(2, 3))
mul = add
add = mul
print(mul(2, 3))
c = 52
```

(a) Original (buggy) program

```python
from operator import (add, mul)
a = add
...
```

(b) PYRITE suggested fix

```python
from operator import (add, mul)
from math import pi
a = add
...
```

(c) Historical student fix

Figure 4.5: An example from the user study where PYRITE performs well.

4.3.4 Threats to validity

As in Chapter 3, the size of our dataset precludes manually annotating the historical programs or our generated repairs. Additionally, we do not have any specifications or test suites with which to test our repaired programs, and we want to count as valid repairs more than just the exact historical repair since guessing that exact repair is frequently both impossible and unnecessary. For example, if a user resolved a type error by replacing a string literal with an integer literal, it would be too high a bar to require PYRITE to guess the exact literal the historical user chose, nor would doing so necessarily provide a better debugging hint than any other literal. We thus consider as our ground truth that a program is repaired if it calls all user-defined functions and does not crash. However this definition of ground truth will in some cases be too permissive, as it can allow for repairs that, for example, prevent a program from crashing by changing a loop condition so the buggy code block within the loop is never executed. We mitigate this threat by also performing an IRB-approved human study where users confirmed that our repairs are frequently *helpful* as well as merely preventing the program from crashing.

For other threats to validity, see Section 3.3.9.

4.4 Related Work

There is a large corpus of related work on program repair.

CLARA [GRZ18] and SARFGEN [WSS18] fix a buggy program by finding previous correct implementations of that same program (for example, correct solutions to the same assignment submitted by other students), and then using those to suggest repairs. Like PYRITE they thus produce repairs which are likely to align with human judgement, but critically PYRITE can use repairs learned from *anywhere* in the dataset, enabling it to repair a new buggy program even if no correct version of that same program exists in the dataset. PYRITE is thus well suited for environments outside the traditional classroom, where there are no other students' submissions to compare to.

TRACER [AKK$^+$18] and RITE [SEC$^+$20] are two of the most similar projects to ours. TRACER uses recurrent neural networks to repair compile-time errors in C programs, and RITE uses deep neural networks to repair (static) type errors in OCaml programs. However, PYRITE is broader and somewhat complementary in scope: it handles a wide variety of runtime errors rather than compile-time errors, it is thus able to obtain and use dynamic features as well as just static ones, and we train and evaluate it on a large and extremely heterogeneous dataset (see Section 3.3.6) instead of on homework assignments to a single class.

HOPPITY [DDL$^+$19] represents Javascript programs as graphs and uses a graph neural network predict a sequence of graph-transformation edits which can repair the program. However, it cannot guarantee that the patched programs it produces do not crash. By using our machine learning models to rank multiple potential fix locations and to produce fix templates instead of concrete fixes, our synthesis step can then continue producing and testing candidate fixes, thus guaranteeing that if it produces a patched program, that program runs without crashing.

GETAFIX [BSPC19] and REVISAR [RSGD18] use a method called anti-unification [KLV14] to find common bug fix templates in a corpus of Java programs. Both these projects are targeted towards the most common categories of Java bugs such as null pointer dereferences and comparing objects by reference instead of value; the fix templates they learn are thus more concrete and apply well to those bug categories but do not handle other kinds of bugs. In contrast, we learn very general syntactic templates and then apply them to generate potential fixes for any kind of runtime error.

PROPHET [LR16] uses a dataset of bug fixes in C programs to learn a probabilistic model which allows it to predict what sequence of pre-determined program transformations will repair a given bug. However, even on programs where PROPHET produces a correct patch at all, it takes on average over two hours to do so. In contrast, we produce our results in under a minute, allowing PYRITE to be used as a component in a realtime debugging workflow.

There is also much related work on producing other kinds of debugging feedback, such as counterexamples - specific inputs which cause the program to exhibit an error. For example, TESTML [SLO19] takes in a reference correct program and a second, incorrect implementation, and uses a combination of enumerative search and symbolic execution to produce as output a concrete example on which the two programs differ. NANOMALY [SJW16] goes further for static type errors, providing as feedback both an example input as well as an interactive execution trace which shows how starting from that input would lead the program to execute an ill-typed expression. While PYRITE shares their overall goal of providing useful debugging feedback to novices, we do so by providing program repairs instead.

4.5 Conclusion

We extend the error localization of PABLO from Chapter 3 into the full-fledged program repair of PYRITE. We use the same set of features but then extend the vectors of buggy program terms with a fix template representing how that term should be repaired. We train a second machine learning model to predict fix templates that will match human judgements for a new buggy program, and then an enumerative synthesis procedure to realize those templates into concrete fixes.

We evaluate our approach on historical bugs from PythonTutor. PYRITE is able to synthesize a repair for 76% of the buggy programs in our dataset. Finally, a user study demonstrates that our approach frequently provides useful debugging information beyond what can be gleaned from the Python interpreter alone.

Acknowledgements for Chapter 4

This chapter describes unpublished work done in collaboration with Madeline Endres, Georgios Sakkas, Westley Weimer, and Ranjit Jhala. The dissertation author was the primary investigator and author of this material.

Appendix A

Dead ends: Adversarial examples

As an alternative to the approach described in Chapter 4, we also attempted to produce repairs by using adversarial examples. Adversarial examples are an idea from machine learning where given a trained classifier and an input point, one finds a new point that is as similar to the input as possible while getting classified differently. For example, in Figure A.1 (taken from [BMR$^+$18]), a small part of a scene is changed to trick a classifier into classifying a banana as a toaster.

Given an 'ideal' classifier, the easiest way to make it classify the image as a toaster is to modify the image until it in fact depicts some kind of toaster. Similarly, we had hypothesized that if we gave existing adversarial-examples tools our PABLO classifier and a vector that PABLO classifies as buggy, the closest vector that the classifier would classify as not buggy might be one that we could turn into a new program term that could actually repair the program.

However in practice, the vectors produced were primarily either nonsense or included non-actionable changes. An example of nonsense would be a vector which breaks an implicit constraint - for example, the type of a program term is represented in our one-hot representation as a sequence of bits - `Is-String`, `Is-Int`, etc - and when we produce such a vector, only one of those bits is a 1. However, the adversarial example generator could make more than one (or

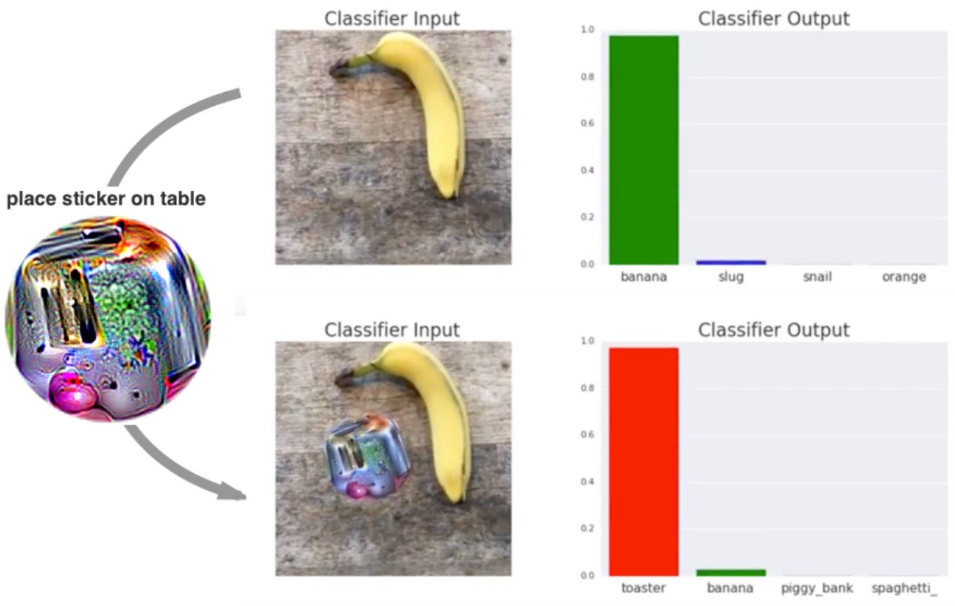


Figure A.1: [Figure taken from [BMR+18]] An example adversarial attack: modifying a small portion of the scene can convince a classifier that this banana is a toaster.

none of them) a 1. Non-actionable changes include global properties encoded in the vector such as the error message. For example, two vectors which differ only in the bits representing the error message ultimately represent the same program term, so it is not useful for the adversarial example generator to tell us to switch from one to the other.

In conclusion this method appeared to be a dead end.

Appendix B

Machine Learning Features

In this appendix we go into further detail about the features used by our machine learning models. See Section 3.2.2 for an overview.

B.1 Global features

Features in this section are computed from the program as a whole, and then that value is used for each vector derived from that program.

B.1.1 Global syntactic features

These 39 boolean features are each set to true if the corresponding syntactic form occurs anywhere in the program. The 39 forms are: **`Assert`**, **`Assign`**, **`AugmentedAssign`**, **`BinaryOp`**, **`Bool`**, **`Break`**, **`ByteStrings`**, **`Call`**, **`Class`**, **`CondExpr`**, **`Conditional`**, **`Continue`**, **`Dictionary`**, **`Dot`**, **`Ellipsis`**, **`Float`**, **`For`**, **`FromImport`**, **`Fun`**, **`Global`**, **`Imaginary`**, **`Import`**, **`Int`**, **`List`**, **`ListComp`**, **`NonLocal`**, **`None`**, **`Paren`**, **`Pass`**, **`Return`**, **`Set`**, **`SlicedExpr`**, **`StmtExpr`**, **`Strings`**, **`Subscript`**, **`Tuple`**, **`UnaryOp`**, **`Var`**, and **`While`**.

B.1.2 Exception type

This categorical feature encodes the type of exception that was thrown at runtime. For most exceptions, we just used the name of the Python exception; the 21 exceptions that appeared in the dataset were `AssertionError`, `AttributeError`, `error`, `ImportError`, `IndexError`, `JSONDecodeError`, `KeyError`, `LookupError`, `MemoryError`, `ModuleNotFoundError`, `NameError`, `OSError`, `OverflowError`, `RecursionError`, `RuntimeError`, `TypeError`, `UnboundLocalError`, `UnicodeDecodeError`, `UnicodeEncodeError`, `ValueError`, and `ZeroDivisionError`.

For particularly common exceptions which have several different meanings, like `TypeError`, we split the exception into multiple categories based on a regular expression match on the rest of the error message. The categories split in this way were `ValueError`, `AttributeError`, `IndexError` (2 categories each), and `TypeError` (13 categories).

B.1.3 Program size

This numeric feature encodes the total number of vectors we generated for the program, i.e. the number of program terms.

B.2 Local features

These features are computed separately for each program term. Each of these features (other than the child number) also have local-contextual versions representing the node's parent and its first three children.

B.2.1 Child number

This numeric feature encodes the node's position among its siblings, e.g. the `y` in `x+y` would have value 2 because it is the second child of the addition node.

B.2.2 Slice, Crash Location, and Size

See Section 3.2.2.

B.2.3 Dynamic type

This categorical feature represents the dynamic type of the term at runtime; the types used were `int`, `str`, `float`, `bool`, `NoneType`, `list`, `set`, `tuple`, `dict`, `function`, `class`, and `instance`, as well as the three special values mentioned in Section 3.2.2.

B.2.4 Syntactic form

This categorical feature represents the syntactic form of the term. We used as categories all AST node names from the `language-python` Haskell package [Pop], except that for unary and binary operators we instead used the name of the specific operator. The categories appearing in the dataset were: `Break`, `Var`, `Continue`, `Multiply`, `Call`, `Int`, `Minus`, `StmtExpr`, `SlicedExpr`, `Divide`, `Plus`, `Assign`, `Global`, `Assert`, `NotEquals`, `ListComp`, `Fun`, `Exponent`, `ByteStrings`, `Set`, `Dot`, `AugmentedAssign`, `None`, `NonLocal`, `While`, `In`, `IsNot`, `Equality`, `LessThan`, `Import`, `NotIn`, `Pass`, `Xor`, `Return`, `Class`, `Invert`, `Float`, `Tuple`, `GreaterThan`, `Modulo`, `And`, `Strings`, `For`, `GreaterThanEquals`, `Paren`, `Dictionary`, `Not`, `LessThanEquals`, `BinaryAnd`, `CondExpr`, `ShiftLeft`, `BinaryOr`, `FromImport`, `Is`, `MatrixMult`, `FloorDivide`, `Conditional`, `Subscript`, `Bool`, `ShiftRight`, `Imaginary`, `Ellipsis`, `Or`, and `List`.

Appendix C

Human Study Stimuli

C.1 PABLO User Study

Below are the source material for the stimuli used in the PABLO user study. For each stimulus, we have the buggy program itself, the top three lines implicated by PABLO, and the line where the program crashes.

PABLO's top three lines: 6, 4, 5. The line where the program crashes: 6.

```
 1  def findLargestOverlap(target, candidate):
 2      start = candidate[0]
 3      count = 0
 4      for i in range(len(target)):
 5          for i in range(len(candidate)):
 6              while candidate[0 + i] == target[len(target) - i]:
 7                  count += 1
 8          return count
 9
10
11
```

```
12 findLargestOverlap('aaaaaaaa', 'aaaaaaaa') #8
13 findLargestOverlap('abcdefg', 'defgabc') #4
14 findLargestOverlap('', '') #-1
15 findLargestOverlap('aaa', '') #-1
16 findLargestOverlap('abcd','cdef') #-> 2
17 findLargestOverlap('TAGGAG', 'GGTAGA') #-> 1
18 findLargestOverlap('aaaa', 'bbbb') #-> 0
19 findLargestOverlap('', 'hijk') #-> -1
20 findLargestOverlap('abc', 'abcd') #-> -1
```

PABLO's top three lines: 14, 13, 12. The line where the program crashes: 14.

```
1  def isIn(char, aStr):
2      '''
3      char: a single character
4      aStr: an alphabetized string
5
6      returns: True if char is in aStr; False otherwise
7
8      Base cases:
9
10     Recursive case:
11     '''
12     half = int(len(aStr)/2)
13     while half != 1 or half != 0:
14         if char == aStr[half]:
15             return True
16         else:
17             if char > aStr[half]:
18                 aStr = aStr[half: : ]
19                 return isIn(char,aStr)
```

```
20             else:
21                 aStr = aStr[ :half: ]
22                 return isIn(char,aStr)
23     return False
24 x = 'a'
25 y = ''
26 print(isIn(x,y))
```

PABLO's top three lines: 14, 9, 8. The line where the program crashes: 14.

```
 1 alphabet='abcdefghijklmnopqrstuvwxyz'
 2 s='cabcdek'
 3 count=0
 4 k=1
 5 file_1=[]
 6 file_2=[]
 7 for i in range(len(s)):
 8     for n in range(len(alphabet)):
 9         if s[i]==alphabet[n]:
10             saved=n
11             print('saved:',saved )
12             print('i:',i)
13             break
14     if s[i]<s[i+1] and s[i+1]<alphabet[saved+2]:
15         print(s[i])
16     else:
17         print("a")
```

PABLO's top three lines: 9, 11, 7. The line where the program crashes: 9.

```
 1 a=int(input())
 2  b=int(input())
 3  c=int(input())
 4  d=int(input())
 5  for i in range(c,d+1):
 6      print('\t',i,end='')
 7  for f in range(a,b+1):
 8      print('\n',f)
 9      for g in range(a,b+1)*(c,d+1):
10          print('\t',g,end='')
11      print()
```

PABLO's top three lines: 5, 3, 7. The line where the program crashes: 5.

```
1  def print_reverse(values):
2      if len(values) == 1:
3          return values
4      else:
5          return values[-1] + print_reverse(values[-1:])
6
7  print_reverse([1,2,3,4,5])
```

PABLO's top three lines: 7, 2, 6. The line where the program crashes: 7.

```
1  n, k=13,[0,1]
2  for i in range(2,n+1):
3      k.append(k[i-2]+k[i-1])
4      if k[i]>n:
5          break
6  n, k=13,[0,1]
7  [k.append([i-2]+k[i-1]) for i in range(2,n)]
```

PABLO's top three lines: 5, 6, 7. The line where the program crashes: 5.

```
1  line = input("Enter some text: ")
2  char = len(line)
3  amount = line.split()
4  whitespace = amount
5  letters = char - whitespace
6  print ("This line has this many words:",len(amount))
7  print ("These words have this many characters:", letters)
```

PABLO's top three lines: 16, 22, 19. The line where the program crashes: 16.

```
1  def isISBN(code):
2  
3      """
4      Geeft True terug als het argument een string is die een geldige ISBN
           ↪ -10 code
5      bevat. Anders wordt False teruggegeven.
6  
7      >>> isISBN('9971502100')
8      True
9      >>> isISBN('9971502108')
10     False
11     >>> isISBN('53WKEFF2C')
12     False
13     >>> isISBN(4378580136)
14     False
15     """
16     if code[:8].isalpha:
17         return False
18 
19     controle = int(code[0])
```

```
20      for i in range(2,10):
21          controle += i * int(code[i-1])
22      controle %= 11
23
24      return code[9] == controle or (code[9] == 'X' and controle == 10)
25  isISBN(9971502100)
```

PABLO's top three lines: 18, 19, 21. The line where the program crashes: 18.

```
1   def count_values_that_are_keys(d):
2       '''(dict) -> int
3
4       Return the number of values in d that are also keys in d.
5
6       >>> count_values_that_are_keys({1: 2, 2: 3, 3: 3})
7       3
8       >>> count_values_that_are_keys({1: 1})
9       1
10      >>> count_values_that_are_keys({1: 2, 2: 3, 3: 0})
11      2
12      >>> count_values_that_are_keys({1: 2})
13      0
14      '''
15
16      result = 0
17      for k in d:
18          if k == d[:k]:
19              result = result + 1
20
21      return result
22  print(count_values_that_are_keys({1: 2, 2: 3, 3: 0}))
```

PABLO’s top three lines: 9, 11, 8. The line where the program crashes: 9.

```
1  def dict_invert(d):
2      '''
3      d: dict
4      Returns an inverted dictionary according to the instructions above
5      '''
6      #YOUR CODE HERE
7      d2 = {}
8      for k in d:
9          d2[d[k]] = list(k)
10 
11     return d2
12 
13 
14 
15 d = {1:10, 2:20, 3:30}
16 dict_invert(d)
```

PABLO’s top three lines: 10, 8, 7. The line where the program crashes: 10.

```
1  a = [int(i) for i in input().split()]
2  
3  res = []
4  if len(a) == 1:
5      print(a)
6  else:
7      for i in range(len(a)):
8          if i == len(a):
9              res.append(a[i-1] + a[0])
10         res.append(a[i-1] + a[i+1])
```

```
11
12 print(res)
```

PABLO's top three lines: 4, 2, 3. The line where the program crashes: 4.

```
1 f = max
2 max = 3
3 result = f(2, 3, 4)
4 max(1, 2)  # Causes an error
```

PABLO's top three lines: 10, 11, 12. The line where the program crashes: 10.

```
 1 import random
 2
 3 reply = input("Enter three integers: ")
 4 s, n, d = reply.split()
 5 s= int(s)
 6 n = int(n)
 7 d = int(d)
 8
 9 octal = random.seed(s)
10 for s in octal:
11     octal = random.seed(s)
12     print(s)
```

PABLO's top three lines: 32, 33, 30. The line where the program crashes: 32.

```
1 def add_hero():
2     Heros_liste = {}
3     print("Comment s'appelle votre super-heros ?")
4     str(input())
```

```
    a = {1 : '', 2 : '', 3 : '', 4 : '', 5 : ''}
    a[1] = input

    print("Dans quelle ville habite-t-il ?")
    b = str(input())
    a[2] = b
    print("Quel est son numero de telephone ?")
    c = int(input())
    a[3] = c
    print("Quels sont ses super-pouvoirs ? (Entrez 0 pour arreter l'
       ↪ enregistrement)")
    d = []
    while "0" not in d :
        d.append(input())
    d.remove("0")
    a[4] = d
    print("Quelle est sa puissance ?")
    e = int(input())
    if 0 < e < 100 :
        print("Votre super-heros a ete ajoute a l'annuaire !")
        a[5] = e
    else :
        while not 0 < e < 100 :
            print("Vous devez rentrer un nombre entier entre 0 et 100 !")
            print("Quelle est sa puissance ?")
            e = int(input())
        print("Votre super-heros a ete ajoute a l'annuaire !")
        a[5] = e
    Heros_liste[:] = a
add_hero()
```

PABLO's top three lines: 6, 4, 8. The line where the program crashes: 6.

```
1
2  def f(x):
3      if len(x) == 1:
4          return str(x)
5      else:
6          [str(f(x[0]) + f(x[1:len(x)]))]
7
8  print(f("emre"))
```

PABLO's top three lines: 45, 37, 38. The line where the program crashes: 45.

```
1  listaprova = list(["a","b","c","cacca","sartini","pupu","boccino","
       ↪ pasqual"])
2  def enumerate(input_list):
3
4      enumerated_list = list()
5
6      a = 0
7
8      for item in input_list:
9
10         if a <= len(input_list):
11
12             f = (a, item)
13             a = a+1
14
15             enumerated_list.append(f)
16         else:
17             break
18
```

```
19
20
21     return enumerated_list
22
23 def partition (input_list, start, end, pivot_position):
24
25     y = input_list[pivot_position]
26     temporary_list = list()
27     temporary_list2 = list()
28     temporary_list3 = list()
29     if end < len(input_list):
30         for item in input_list[end +1:len(input_list)+ 1]:
31             temporary_list.append(item)
32             input_list.remove(item)
33     for item2 in input_list[start:end+1]:
34         if item2 > y:
35             temporary_list2.append(item2)
36             input_list.remove(item2)
37     temporary_list3.append(y)
38     input_list.remove(y)
39     input_list.extend(temporary_list3)
40     input_list.extend(temporary_list2)
41     input_list.extend(temporary_list)
42     for position, item3 in enumerate(input_list):
43         if item3 == y:
44             return position
45 partition(listaprova)
```

PABLO's top three lines: 3, 2, 1. The line where the program crashes: 3.

```
1 i=2
```

```
2 for i in "1234":
3     print (i*i, end = ' ')
```

PABLO’s top three lines: 9, 8, 2. The line where the program crashes: 9.

```
1  dna = "AGGCtGCctcTgggCGCATtgaTggTtTATTAACGACTAAAcacacac"
2  dna=dna.upper()
3  def firstSSR(dna,seq):
4      if seq not in dna:
5          return 0
6      else:
7         n=1
8         while seq*n in dna:
9             return len(seq*n/2)
10 firstSSR(dna,'AC')
```

PABLO’s top three lines: 9, 7, 5. The line where the program crashes: 9.

```
1 def gcd():
2     if a == b:
3         return b
4     elif a < b:
5             gcd(b, a)
6     else:
7         return gcd(a - b, a)
8
9 gcd(34, 19)
```

PABLO’s top three lines: 12, 11, 9. The line where the program crashes: 12.

```
1 i=[]
```

```
 2 x=int(input())
 3 r=x%2
 4 if (r==0):
 5     x=x+1
 6 i.append(x)
 7 for e in range(5):
 8     x=x+2
 9     i.append(x)
10
11 for e in range(6):
12     print(x[e])
```

PABLO's top three lines: 8, 7, 6. The line where the program crashes: 6.

```
1 from math import pow
2 n=int(input('Ingresa el valor de n:'))
3 suma=0
4 k=0
5 for k in range (0,n+1):
6     suma+=(2*(pow(-1,k))*(pow(3,((1/2)-k)))/(2*k)+1)
7     print(suma)
8     k+=1
```

PABLO's top three lines: 9, 20, 19. The line where the program crashes: 9.

```
1 class Point:
2     def setx(self, x):
3         self.x=x
4
5     def sety(self,y):
6         self.x=y
```

```
 7
 8      def get(self):
 9          return self.x, self.y
10
11      def move(self, dx,dy):
12          self.x=self.x+dx
13          self.y=self.y+dy
14
15
16  p=Point()
17  p.setx(3)
18  p.sety(4)
19  print(p.get())
20  p.move(1,3)
```

PABLO's top three lines: 8, 15, 12. The line where the program crashes: 8.

```
 1  total = 0
 2  items = ""
 3
 4  def adding_report(report = "T"):
 5      while True:
 6          tally = input("Input an integer to add to the total or \"Q\" to
               ↪ quit.")
 7          if tally.isdigit():
 8              tally + total
 9              items = print(tally)
10              input("Enter an integer or \"Q\":")
11          elif tally.lower() == "quit":
12              print("This is your total: " + total)
13              break
```

```
14         else:
15             print()
16             break
17
18 adding_report()
```

PABLO's top three lines: 4, 2, 5. The line where the program crashes: 4.

```
1 import math
2 m = input("Please give me give a mass (in kilograms): ")
3 e = 0
4 e = (m * 299792458 ** 2)
5 print("The equivalent energy is", e, "Joules")
```

PABLO's top three lines: 5, 1, 4. The line where the program crashes: 5.

```
1 name = "pedro eaeae"
2
3 print(name.upper())
4 print(name.lower())
5 print(name.rstring())
```

PABLO's top three lines: 6, 7, 9. The line where the program crashes: 6.

```
1 x = "carlos90"
2 r=list(x)
3 print(r)
4 d=list()
5 for i in x:
6     s=int(i)
7     d.append(s)
```

```
8
9 print(d)
```

PABLO's top three lines: 15, 8, 14. The line where the program crashes: 15.

```
1  class Domino():
2
3      def __init__(self, a=0,b=0):
4          self.A=a
5          self.B=b
6      def affiche_points(self):
7          print("face A: "+str(self.A)+"  "+"face B: "+str(self.B))
8      def valeur(self):
9          d=self.A+self.B
10
11 d1 = Domino(2,6)
12 d2 = Domino(4,3)
13 d1.affiche_points()
14 d2.affiche_points()
15 print("total des points :", d1.valeur() + d2.valeur())
```

PABLO's top three lines: 3, 8, 4. The line where the program crashes: 3.

```
1 a = [int(i) for i in input().split()]
2 m = 0
3 for i in range(a):
4     if a[m]%2 == 0:
5         a[m] //= 2
6         m -= 1
7     else:
8         del a[m]
```

```
 9
10 print(a)
```

PABLO's top three lines: 8, 5, 10. The line where the program crashes: 8.

```
 1 def threshold_prices(prices):
 2
 3     threshold_val = 40
 4
 5     for x in prices:
 6
 7         if x > threshold_val:
 8             return 'There is at least one price over' + threshold_val
 9
10     return 'There is no prices over'+ threshold_val
11
12 print(threshold_prices([20, 40, 50]))
```

PABLO's top three lines: 9, 2, 1. The line where the program crashes: 9.

```
1 firstNumber=float(input("enter first number"))
2 secondNumber=float(input("enter second number"))
3 print("sum:" ,  firstNumber+secondNumber )
4 print("difference:" ,  firstNumber-secondNumber )
5 print("product:" ,  firstNumber*secondNumber )
6 print("average:" ,  firstNumber+secondNumber/2 )
7 print("maximum:" ,  max(firstNumber,secondNumber) )
8 print("minimum:" ,  min(firstNumber,secondNumber ))
9 print("distance" ,  abs(firstNumber,secondNumber))
```

PABLO's top three lines: 13, 6, 3. The line where the program crashes: 13.

```python

#Get a character to represent the first allele
D_allele = input("Enter a character for the first allele: ")

#get the p frequency
p_freq = input("Enter the frequency of " + D_allele + " (between 0 - 1):
    ↪ ")

#Get the second allele character (q)
r_allele = input("Enter a different character for the second allele: ")
print()

#Calculate the q frequency, round the result to 2 decimal places
q_freq = round(1 - p_freq,2)

#Print out the frequencies of the alleles
print("If the frequency of ", D_allele," in the population is: ", p_freq)
print("Then, the frequency of ", r_allele," in the population is: ",
    ↪ q_freq)
print()
```

C.2 PYRITE User Study

Below are the source material for the stimuli used in the PYRITE user study. For each stimulus, we have the buggy program itself, any input provided to the program (one line per call to `input()`), the novice's historical fix, and PYRITE's suggested repair.

Buggy Python Program #1

```
speed = input("what was your average speed (in mph)?:   ")
time = input("what was the time you took (in min)?:   ")
print("the distance you travelled is (in miles):   ")
print(speed * time)
```

Program Input

```
10
10
```

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 4, in <module>
    print(speed * time)
TypeError: can't multiply sequence by non-int of type 'str'
```

Pyrite Fix

```
speed = input("what was your average speed (in mph)?:   ")
time = input("what was the time you took (in min)?:   ")
print("the distance you travelled is (in miles):   ")
print(speed)
```

Historical Fix

```
speed = int(input("what was your average speed (in mph)?:   "))
time = int(input("what was the time you took (in min)?:   "))
print("the distance you travelled is (in miles):   ")
print(speed * time)
```

Buggy Python Program #2

```
x = int(input("Digite um numero: "))
i = 1
k = 1
```

```
b = 4
while k <= x:
    a = 2
    while a < b:
        if b % a == 0:
            b += 1
            z = 0
            break
        else:
            a += 1
            z = 1
    if z == 1:
        print(b)
        soma = soma + b
        k += 1
print("a soma dos %d primeiros numeros = %d", x, soma)
```

Program Input

```
5
```

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 17, in <module>
    soma = soma + b
NameError: name 'soma' is not defined
```

Pyrite Fix

```
x = int(input("Digite um numero: "))
i = 1
k = 1
b = 4
soma = 1
while k <= x:
    a = 2
    while a < b:
        if b % a == 0:
            b += 1
            z = 0
            break
        else:
            a += 1
            z = 1
    if z == 1:
        print(b)
        soma = soma + b
        k += 1
print("a soma dos %d primeiros numeros = %d", x, soma)
```

Historical Fix

```
x = int(input("Digite um numero: "))
i = 1
k = 1
b = 4
soma = 0
while k <= x:
    a = 2
    while a < b:
        if b % a == 0:
            b += 1
            z = 0
            break
        else:
            a += 1
            z = 1
    if z == 1:
        print(b)
        soma = soma + b
        k += 1
print("a soma dos %d primeiros numeros = %d", x, soma)
```

Buggy Python Program #3

```
a = [1, 1]
b = [1, 1]
c = a + [b]
c = c[1:2]
while a:
    b.extend([[a.pop()]])
    d, b = b, d
a = b
b[2][0], d = c, b[2]
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File “File_Path/buggy_code.py”, line 7, in <module>
    d, b = b, d
NameError: name ’d’ is not defined
```

Pyrite Fix

```
a = [1, 1]
b = [1, 1]
c = a + [b]
c = c[1:2]
while a:
    b.extend([[a.pop()]])
    d, b = b, b
a = b
b[2][0], d = c, b[2]
```

Historical Fix

```
a = [1, 1]
b = [1, 1]
c = a + [b]
d = c[1:2]
while a:
    b.extend([[a.pop()]])
    d, b = b, d
a = b
b[2][0], d = c, b[2]
```

Buggy Python Program #4

```
def is_anagram(test, original):
    test = test.lower()
    SortedTest = ''.join(sorted(test))
    print(SortedTest)
    original = original.lower()
    SortedOriginal = ''.join(sorted(original))
    print(SortedOriginal)
    if SortedTest == SortedOriginal:
        Print('True')
is_anagram("Twoo", "WooT")
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 10, in <module>
    is_anagram("Twoo", "WooT")
  File "File_Path/buggy_code.py", line 9, in is_anagram
    Print('True')
NameError: name 'Print' is not defined
```

Pyrite Fix and Historical Fix

```
def is_anagram(test, original):
    test = test.lower()
    SortedTest = ''.join(sorted(test))
    print(SortedTest)
    original = original.lower()
    SortedOriginal = ''.join(sorted(original))
    print(SortedOriginal)
    if SortedTest == SortedOriginal:
        print('True')
is_anagram("Twoo", "WooT")
```

Buggy Python Program #5

```
encrypt = ’E’
if encrypt == ’E’:
    supp_id = input(“Enter the supplier id:”)
    supp_id_str = str(supp_id)
    length = len(supp_id_str)
    supp = [“”]
    list1 = [length, “-”, supp_id]
    supp_id_str = supp_id_str + (str(length))
    supp_id_rev = supp_id_str.reverse()
    x = 0
    a = len(list1)
    for x in range(a):
        for i in list1:
            if not supp:
                supp = i
            else:
                supp.append([i])
    print(i)
```

Program Input

```
13
```

Python Error Message

```
Traceback (most recent call last):
  File “File_Path/buggy_code.py”, line 9, in <module>
    supp_id_rev = supp_id_str.reverse()
AttributeError: ’str’ object has no attribute ’reverse’
```

Pyrite Fix

```
encrypt = ’E’
if encrypt == ’E’:
    supp_id = input(“Enter the supplier id:”)
    supp_id_str = str(supp_id)
    length = len(supp_id_str)
    supp = [“”]
    list1 = [length, “-”, supp_id]
    supp_id_str = supp_id_str + (str(length))
    supp_id_rev = supp_id_str
    x = 0
    a = len(list1)
    for x in range(a):
        for i in list1:
            if not supp:
                supp = i
            else:
                supp.append([i])
    print(i)
```

Historical Fix

```
encrypt = 'E'
if encrypt == 'E':
    supp_id = input("Enter the supplier id:")
    supp_id_str = str(supp_id)
    length = len(supp_id_str)
    supp = [""]
    list1 = [length, "-", supp_id]
    supp_id_str = supp_id_str + (str(length))
    supp_id_rev = supp_id_str[::- 1]
    x = 0
    a = len(list1)
    for x in range(a):
        for i in list1:
            if not supp:
                supp = i
            else:
                supp.append([i])
    print(i)
```

Buggy Python Program #6

```
text = str("32 apples")
for i in range(len(s)):
    char = s[i][0]
    if char.isdigit():
        print(i)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    for i in range(len(s)):
NameError: name 's' is not defined
```

Pyrite Fix

```
text = str("32 apples")
for i in range(len('')):
    char = s[i][0]
    if char.isdigit():
        print(i)
```

Historical Fix

```
text = str("32 apples")
for i in range(len(text)):
    char = text[i][0]
    if char.isdigit():
        print(i)
```

Buggy Python Program #7

```
text = 'ahoj'
for i in range(4):
    if i > len(text):
        print('*')
    medzery = 3
    for j in range(i, i + 13, 4):
        if j > len(text):
            print('*' * medzery)
        else:
            print(''.format(text[j]), end="")
            medzery -= 1
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 10, in <module>
    print(''.format(text[j]), end="")
IndexError: string index out of range
```

Pyrite Fix

```
text = 'ahoj'
for i in range(4):
    if i > len(text):
        print('*')
    medzery = 3
    for j in range(i, i + 1, 4):
        if j > len(text):
            print('*' * medzery)
        else:
            print(''.format(text[j]), end="")
            medzery -= 1
```

Historical Fix

```
text = 'ahoj'
for i in range(4):
    if i > len(text):
        print('*')
    medzery = 3
    for j in range(i, i + 13, 4):
        if j >= len(text):
            print('*' * medzery)
        if j < len(text):
            print(''.format(text[j]), end="")
            medzery -= 1
```

Buggy Python Program #8

```
def has_sum(total, n1, n2):
    if total - n1 < 0 or total - n2 < 0:
        return false
    if total - n1 == 0:
        return true
    elif total - n2 == 0:
        return true
    else:
        has_sum(total - n1, n1, n2) or has_sum(total - n2, n1, n2)
        return false
has_sum(1, 3, 5)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 11, in <module>
    has_sum(1, 3, 5)
  File "File_Path/buggy_code.py", line 3, in has_sum
    return false
NameError: name 'false' is not defined
```

Pyrite Fix

```
def has_sum(total, n1, n2):
    if total - n1 < 0 or total - n2 < 0:
        return 0
    if total - n1 == 0:
        return true
    elif total - n2 == 0:
        return true
    else:
        has_sum(total - n1, n1, n2) or has_sum(total - n2, n1, n2)
        return false
has_sum(1, 3, 5)
```

Historical Fix

```
def has_sum(total, n1, n2):
    if total - n1 < 0 or total - n2 < 0:
        return False
    if total - n1 == 0:
        return True
    elif total - n2 == 0:
        return True
    else:
        has_sum(total - n1, n1, n2) or has_sum(total - n2, n1, n2)
        return False
has_sum(1, 3, 5)
```

Buggy Python Program #9

```
sub(5, 2)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 1, in <module>
    sub(5, 2)
NameError: name 'sub' is not defined
```

Pyrite Fix

```
range(5, 2)
```

Historical Fix

```
from operator import *
sub(5, 2)
```

Buggy Python Program #10

```
cats = [1, 2]
dogs = [cats, cats.append(23), list(cast)]
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    dogs = [cats, cats.append(23), list(cast)]
NameError: name 'cast' is not defined
```

Pyrite Fix and Historical Fix

```
cats = [1, 2]
dogs = [cats, cats.append(23), list(cats)]
```

Buggy Python Program #11

```
def specialSum(n):
    theSum = 0
    for i in range(1, n):
        if i % 2 == 0 or i % 3 == 0:
            if i % 5 == 1:
                theSum += 1
    return theSum
specialSum(10)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 8, in <module>
    specialSum(10)
  File "File_Path/buggy_code.py", line 6, in specialSum
```

```
    theSum += l
NameError: name 'l' is not defined
```

Pyrite Fix

```
def specialSum(n):
    theSum = 0
    for i in range(1, n):
        if i % 2 == 0 or i % 3 == 0:
            if i % 5 == 1:
                theSum += 0
    return theSum
specialSum(10)
```

Historical Fix

```
def specialSum(n):
    theSum = 0
    for i in range(1, n):
        if i % 2 == 0 or i % 3 == 0:
            if i % 5 == 1:
                theSum += i
    return theSum
specialSum(10)
```

Buggy Python Program #12

```
a = 'AUG'
b = 'AUG': 'Met', 'GAG': 'Glu'
print(dict.get(a))
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 3, in <module>
    print(dict.get(a))
TypeError: descriptor 'get' requires a 'dict' object but received a 'str'
```

Pyrite Fix and Historical Fix

```
a = 'AUG'
b = 'AUG': 'Met', 'GAG': 'Glu'
print(b.get(a))
```

Buggy Python Program #13

```
def mergeSort(nums):
    n = len(nums)
    if n > 1:
        m = n // 2
        nums1, nums2 = nums[:m], nums[m:]
        mergeSort(nums1)
        mergeSort(nums2)
        merge(nums1, nums2, nums)
def merge(lst1, lst2, lst3):
    i1 = i2 = i3 = 0
    n1, n2 = len(lst1), len(lst2)
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]:
            lst3[i3] = lst1[i1]
            i1 = i1 + 1
        else:
            lst3[i3] = lst2[i2]
            i2 = i2 + 1
            i3 = i3 + 1
    while i1 < len(lst1):
        lst3[i3] = lst1[i1]
        i1 = i1 + 1
        i3 = i3 + 1
    while i2 < len(lst2):
        lst3[i3] = lst2[i2]
        i2 = i2 + 1
        i3 = i3 + 1
MergeSort([1, 5, 2, 5, 6, 11])
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File “File_Path/buggy_code.py”, line 28, in <module>
    MergeSort([1, 5, 2, 5, 6, 11])
NameError: name ’MergeSort’ is not defined
```

Pyrite Fix and Historical Fix

```
def mergeSort(nums):
    n = len(nums)
    if n > 1:
        m = n // 2
        nums1, nums2 = nums[:m], nums[m:]
        mergeSort(nums1)
        mergeSort(nums2)
        merge(nums1, nums2, nums)
def merge(lst1, lst2, lst3):
    i1 = i2 = i3 = 0
    n1, n2 = len(lst1), len(lst2)
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]:
            lst3[i3] = lst1[i1]
            i1 = i1 + 1
        else:
            lst3[i3] = lst2[i2]
            i2 = i2 + 1
            i3 = i3 + 1
    while i1 < len(lst1):
        lst3[i3] = lst1[i1]
        i1 = i1 + 1
        i3 = i3 + 1
    while i2 < len(lst2):
        lst3[i3] = lst2[i2]
        i2 = i2 + 1
        i3 = i3 + 1
mergeSort([1, 5, 2, 5, 6, 11])
```

Buggy Python Program #14

```
x = [’-1’, ’-2’, ’-3’, ’-4’][3]
numbers = []
while x < - 1:
    z = - x
    numbers.append(z)
    x = x + 1
print(numbers)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File “File_Path/buggy_code.py”, line 3, in <module>
    while x < - 1:
TypeError: ’<’ not supported between instances of ’str’ and ’int’
```

Pyrite Fix

```
x = [’-1’, ’-2’, ’-3’, ’-4’][3]
numbers = []
while len(x) < - 1:
    z = - x
    numbers.append(z)
    x = x + 1
print(numbers)
```

Historical Fix

```
x = [’-1’, ’-2’, ’-3’, -_4][3]
numbers = []
while x < - 1:
    z = - x
    numbers.append(z)
    x = x + 1
print(numbers)
```

Buggy Python Program #15

```
mystr = 'dog'
print(mystr.startswith(0, 3))
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    print(mystr.startswith(0, 3))
TypeError: startswith first arg must be str or a tuple of str, not int
```

Pyrite Fix

```
mystr = 'dog'
print(mystr.startswith('dog', 3))
```

Historical Fix

```
mystr = 'dog'
print(mystr.startswith('d'))
```

Buggy Python Program #16

```
a = [1, 2, 3]
b = list(b)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    b = list(b)
NameError: name 'b' is not defined
```

Pyrite Fix and Historical Fix

```
a = [1, 2, 3]
b = list(a)
```

Buggy Python Program #17

```
S = str(input())
lst = S.split(_)
print(lst)
```

Program Input

```
my_first_class
```

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    lst = S.split(_)
NameError: name '_' is not defined
```

Pyrite Fix

```
S = str(input())
lst = S.split('a')
print(lst)
```

Historical Fix

```
S = str(input())
lst = S.split("_")
print(lst)
```

Buggy Python Program #18

```
from operator import (add, mul)
from math import c
a = add
b = mul
print(mul(2, 3))
mul = add
add = mul
print(mul(2, 3))
c = 52
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    from math import c
ImportError: cannot import name 'c' from 'math'
```

Pyrite Fix

```
from operator import (add, mul)
a = add
b = mul
print(mul(2, 3))
mul = add
add = mul
print(mul(2, 3))
c = 52
```

Historical Fix

```
from operator import (add, mul)
from math import pi
a = add
b = mul
print(mul(2, 3))
mul = add
add = mul
```

```
print(mul(2, 3))
c = 52
```

Buggy Python Program #19

```
def greedySum(L, s):
    "”" input: s, positive integer, what the sum should add up to
            L, list of unique positive integers sorted in descending order
    Use the greedy approach where you find the largest multiplier for
    the largest value in L then for the second largest, and so on to
    solve the equation s = L[0]*m_0 + L[1]*m_1 + ... + L[n-1]*m_(n-1)
    return: the sum of the multipliers or "no solution” if greedy approach
    does not yield a set of multipliers such that the equation sums to ’s’
    "”"
    outputList = []
    sumOf = []
    tempS = s
    for n in L:
        print(’n in L’, n, ’s’, tempS, ’s//n’, tempS // n)
        if tempS / n >= 1:
            check = tempS // n
            outputList.append(check)
            sumOf.append(n * check)
            tempS -= n
        print(outputList, sumOf, sum(sumOf))
    print(sum(sumOf), s)
    if sum(sumOf) == s:
        return sum(multiList)
    else:
        return ’no solution’
L = [10, 5, 1]
s = 11
print(greedySum(L, s))
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 28, in <module>
    print(greedySum(L, s))
  File "File_Path/buggy_code.py", line 23, in greedySum
    return sum(multiList)
NameError: name 'multiList' is not defined
```

Pyrite Fix

```
def greedySum(L, s):
    """ input: s, positive integer, what the sum should add up to
    L, list of unique positive integers sorted in descending order
    Use the greedy approach where you find the largest multiplier for
    the largest value in L then for the second largest, and so on to
    solve the equation s = L[0]*m_0 + L[1]*m_1 + ... + L[n-1]*m_(n-1)
    return: the sum of the multipliers or "no solution" if greedy approach
    does not yield a set of multipliers such that the equation sums to 's'
    """
    outputList = []
    sumOf = []
    tempS = s
    for n in L:
        print('n in L', n, 's', tempS, 's//n', tempS // n)
        if tempS / n >= 1:
            check = tempS // n
            outputList.append(check)
            sumOf.append(n * check)
            tempS -= n
        print(outputList, sumOf, sum(sumOf))
    print(sum(sumOf), s)
    if sum(sumOf) == s:
        return sum(L)
    else:
        return 'no solution'
L = [10, 5, 1]
s = 11
print(greedySum(L, s))
```

Historical Fix

```
def greedySum(L, s):
    """ input: s, positive integer, what the sum should add up to
    L, list of unique positive integers sorted in descending order
    Use the greedy approach where you find the largest multiplier for
    the largest value in L then for the second largest, and so on to
    solve the equation s = L[0]*m_0 + L[1]*m_1 + ... + L[n-1]*m_(n-1)
    return: the sum of the multipliers or "no solution" if greedy approach
    does not yield a set of multipliers such that the equation sums to 's'
    """
    outputList = []
    sumOf = []
    tempS = s
    for n in L:
        print('n in L', n, 's', tempS, 's//n', tempS // n)
        if tempS / n >= 1:
            check = tempS // n
            outputList.append(check)
            sumOf.append(n * check)
            tempS -= n
        print(outputList, sumOf, sum(sumOf))
    print(sum(sumOf), s)
    if sum(sumOf) == s:
        return sum(outputList)
    else:
        return 'no solution'
L = [10, 5, 1]
s = 11
print(greedySum(L, s))
```

Buggy Python Program #20

```
counter = ['this', 'is', 'spot', 'see', 'spot', 'run']
count =
for item in counter:
    self.count[item] = self.count.get(item, 0) + 1
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 4, in <module>
    self.count[item] = self.count.get(item, 0) + 1
NameError: name 'self' is not defined
```

Pyrite Fix

```
counter = ['this', 'is', 'spot', 'see', 'spot', 'run']
count =
counter = ''
for item in counter:
    self.count[item] = self.count.get(item, 0) + 1
```

Historical Fix

```
counter = ['this', 'is', 'spot', 'see', 'spot', 'run']
count =
for item in counter:
    count[item] = count.get(item, 0) + 1
```

Buggy Python Program #21

```
def interval(a, b):
    """Construct an interval from a to b."""
    return [a, b]
def str_interval(x):
    """Return a string representation of interval x."""
    return '0 to 1'.format(lower_bound(x), upper_bound(x))
str_interval(interval(- 1, 2))
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 7, in <module>
    str_interval(interval(- 1, 2))
  File "File_Path/buggy_code.py", line 6, in str_interval
    return '0 to 1'.format(lower_bound(x), upper_bound(x))
NameError: name 'lower_bound' is not defined
```

Pyrite Fix

```
def interval(a, b):
    """Construct an interval from a to b."""
    return [a, b]
def str_interval(x):
    """Return a string representation of interval x."""
    return '0 to 1'.format(input(x), upper_bound(x))
str_interval(interval(- 1, 2))
```

Historical Fix

```
def lower_bound(x):
    """Return the lower bound of interval x."""
    "*** YOUR CODE HERE ***"
    return min(x)
def upper_bound(x):
    """Return the upper bound of interval x."""
    "*** YOUR CODE HERE ***"
def interval(a, b):
    """Construct an interval from a to b."""
    return [a, b]
def str_interval(x):
    """Return a string representation of interval x."""
    return '0 to 1'.format(lower_bound(x), upper_bound(x))
str_interval(interval(- 1, 2))
```

Buggy Python Program #22

```
def inc(binary):
    reverse_bin = binary[::- 1]
    final_bin = “”
    for bit in reverse_bin:
        if bit == “1”:
            final_bin += “0”
        else:
            final_bin += “1”
            final_bin += reverse_bin[reverse.find(bit) + 1:]
            return final_bin[::- 1]
    return (“1” + final_bin)
inc(“1110011”)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File “File_Path/buggy_code.py”, line 12, in <module>
    inc(“1110011”)
  File “File_Path/buggy_code.py”, line 9, in inc
    final_bin += reverse_bin[reverse.find(bit) + 1:]
NameError: name ’reverse’ is not defined
```

Pyrite Fix

```
def inc(binary):
    reverse_bin = binary[::- 1]
    final_bin = “”
    for bit in reverse_bin:
        if bit == “1”:
            final_bin += “0”
        else:
            final_bin += “1”
            final_bin += reverse_bin[binary.find(bit) + 1:]
            return final_bin[::- 1]
    return (“1” + final_bin)
inc(“1110011”)
```

Historical Fix

```
def inc(binary):
    reverse_bin = binary[::- 1]
    final_bin = “”
    for bit in reverse_bin:
        if bit == “1”:
            final_bin += “0”
        else:
            final_bin += “1”
            final_bin += reverse_bin[reverse_bin.find(bit) + 1:]
            return final_bin[::- 1]
    return (“1” + final_bin)
inc(“1110011”)
```

Buggy Python Program #23

```
def pingpong(n):
    def helper_function(counter, flag):
        if counter % 7 == 0 or has_seven(counter):
            flag = not flag
        return flag
    if n == 1:
        return n
    if helper_function(n - 1, True):
        return pingpong(n - 1) + 1
    else:
        return pingpong(n - 1) - 1
pingpong(9)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File “File_Path/buggy_code.py”, line 12, in <module>
    pingpong(9)
  File “File_Path/buggy_code.py”, line 8, in pingpong
    if helper_function(n - 1, True):
```

```
  File "File_Path/buggy_code.py", line 3, in helper_function
    if counter % 7 == 0 or has_seven(counter):
NameError: name 'has_seven' is not defined
```

Pyrite Fix

```
def pingpong(n):
    def helper_function(counter, flag):
        if counter % 7 == 0 or range(counter):
            flag = not flag
        return flag
    if n == 1:
        return n
    if helper_function(n - 1, True):
        return pingpong(n - 1) + 1
    else:
        return pingpong(n - 1) - 1
pingpong(9)
```

Historical Fix

```
def has_seven(k):
    if k % 10 == 7:
        return True
    elif k < 10:
        return False
    else:
        return has_seven(k // 10)
def pingpong(n):
    def helper_function(counter, flag):
        if counter % 7 == 0 or has_seven(counter):
            flag = not flag
        return flag
    if n == 1:
        return n
    if helper_function(n - 1, True):
        return pingpong(n - 1) + 1
    else:
        return pingpong(n - 1) - 1
pingpong(9)
```

Buggy Python Program #24

```
print(accumulate([1, 2, 3, 4, 5]))
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 1, in <module>
    print(accumulate([1, 2, 3, 4, 5]))
NameError: name 'accumulate' is not defined
```

Pyrite Fix

```
print(print([1, 2, 3, 4, 5]))
```

Historical Fix

```
from itertools import accumulate
print(accumulate([1, 2, 3, 4, 5]))
```

Buggy Python Program #25

```
n = [3, 2, 4]
target = 6
for n in nums:
    for m in nums:
        if n + m == target:
            print("[,]".format(nums.index(n), nums.index(m)))
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 3, in <module>
    for n in nums:
NameError: name 'nums' is not defined
```

Pyrite Fix

```
n = [3, 2, 4]
target = 6
for n in '':
    for m in nums:
        if n + m == target:
            print("[,]".format(nums.index(n), nums.index(m)))
```

Historical Fix

```
nums = [3, 2, 4]
target = 6
for n in nums:
    for m in nums:
        if n + m == target:
            print("[,]".format(nums.index(n), nums.index(m)))
```

Buggy Python Program #26

```
s = 'abcde'
char = s(0)
print(char)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 2, in <module>
    char = s(0)
TypeError: 'str' object is not callable
```

Pyrite Fix

```
s = 'abcde'
char = s[0]
print(char)
```

Historical Fix

```
s = 'abcde'
char = s[0]
```

Buggy Python Program #27

```
def shell_sort(Array):
    N = len(Array)
    h = 1
    while h <= N // 3:
        h = 3 * h + 1
    while h >= 1:
        for i in range(h, N):
            j = i
            while j >= h and Array[j] < Array[j - h]:
                print(Array[j])
                Array[j], Array[j - h] = Array[j - h], Array[j]
                j -= h
        h = h / 3
    return Array
def main():
    x = [3, 2, 1, 4, 5, 4, 1, 4, 5, 6, 89, 9, 22]
    print(shell_sort(x))
main()
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 18, in <module>
    main()
  File "File_Path/buggy_code.py", line 17, in main
    print(shell_sort(x))
  File "File_Path/buggy_code.py", line 7, in shell_sort
    for i in range(h, N):
TypeError: 'float' object cannot be interpreted as an integer
```

Pyrite Fix

```
def shell_sort(Array):
    N = len(Array)
    h = 1
    while h <= h // 3:
        h = 3 * h + 1
    while h >= 1:
        for i in range(h, N):
            j = i
            while j >= h and Array[j] < Array[j - h]:
                print(Array[j])
                Array[j], Array[j - h] = Array[j - h], Array[j]
                j -= h
        h = h / 3
    return Array
def main():
    x = [3, 2, 1, 4, 5, 4, 1, 4, 5, 6, 89, 9, 22]
    print(shell_sort(x))
main()
```

Historical Fix

```
def shell_sort(Array):
    N = len(Array)
    h = 1
    while h <= N // 3:
        h = 3 * h + 1
    while h >= 1:
        for i in range(h, N):
            j = i
            while j >= h and Array[j] < Array[j - h]:
                print(Array[j])
                Array[j], Array[j - h] = Array[j - h], Array[j]
                j -= h
        h = h // 3
    return Array
def main():
    x = [3, 2, 1, 4, 5, 4, 1, 4, 5, 6, 89, 9, 22]
    print(shell_sort(x))
main()
```

Buggy Python Program #28

```
def f(y):
    print(x)
def g():
    global x
    print(x)
    x = 'g'
x = 'global_string'
f()
g()
f(5)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 8, in <module>
    f()
TypeError: f() missing 1 required positional argument: 'y'
```

Pyrite Fix

```
def f(y):
    print(x)
def g():
    global x
    print(x)
    x = 'g'
x = 'global_string'
g()
g()
f(5)
```

Historical Fix

```
def f():
    print(x)
def g():
    global x
    print(x)
    x = 'g'
x = 'global_string'
f()
g()
f()
```

Buggy Python Program #29

```
message = 'It was a bright cold day in April.'
count = 
for character in message:
    count.setdefault(character)
    count[character] = count[character] + 1
print(count)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 5, in <module>
    count[character] = count[character] + 1
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Pyrite Fix

```
message = ''
count = 
for character in message:
    count.setdefault(character)
    count[character] = count[character] + 1
print(count)
```

Historical Fix

```
message = 'It was a bright cold day in April.'
count = 
for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1
print(count)
```

Buggy Python Program #30

```
def float_range(start, stop, skip):
    yo = [start]
    i = start
    while i < stop:
        i += skip
        if i == stop:
            yo.remove(i)
        else:
            yo.append(i)
    return yo
float_range(0, 8, 1)
```

(No program input provided)

Python Error Message

```
Traceback (most recent call last):
  File "File_Path/buggy_code.py", line 11, in <module>
    float_range(0, 8, 1)
  File "File_Path/buggy_code.py", line 7, in float_range
    yo.remove(i)
ValueError: list.remove(x): x not in list
```

Pyrite Fix

```
def float_range(start, stop, skip):
    yo = [start]
    i = start
    while i < stop:
        i += skip
        if i == stop:
            yo.remove(i)
        else:
            yo.append(8)
    return yo
float_range(0, 8, 1)
```

Historical Fix

```
def float_range(start, stop, skip):
    yo = [start]
    i = start
    while i < stop:
        i += skip
        if i == stop:
            yo.pop()
        else:
            yo.append(i)
    return yo
float_range(0, 8, 1)
```

Bibliography

[ACFH11] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *POPL*, 2011.

[ADG$^+$13] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1976–1982. AAAI Press, 2013.

[AKK$^+$18] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: For the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, page 78–87, New York, NY, USA, 2018. Association for Computing Machinery.

[AL80] Anne Adam and Jean-Pierre Laurent. Laura, a system to debug student programs. *Artificial Intelligence*, 15(1):75 – 122, 1980.

[AP10] Nathaniel Ayewah and William Pugh. The Google Findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, 2010.

[AZG06] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[AZGvG09] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BBC$^+$10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[BDB] bdb Debugger Framework. `https://docs.python.org/2/library/bdb.html`.

[BLLLGG16] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, 2016.

[BMR$^+$18] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch, 2018.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Bre17] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[BSPC19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability modulo theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. 1 edition, 2009.

[Bü60] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[Cau] Dino Causevic. Structured tree comparison with tree kernels. `https://www.toptal.com/machine-learning/structured-data-tree-kernels`. Accessed: 2018-08-21.

[CE14] Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM.

[CES$^+$20] Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer. Pablo: Helping novices debug python code through data-driven fault localization. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 1047–1053, New York, NY, USA, 2020. Association for Computing Machinery.

[Chr14] David Raymond Christiansen. Reflect on your mistakes! lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, 2014.

[CKF$^+$02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[CZ05] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[DDL$^+$19] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2019.

[DKA$^+$15] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. How can automatic feedback help students construct automata? *ACM Trans. Comput.-Hum. Interact.*, 22(2):9:1–9:24, March 2015.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1859–1866, New York, NY, USA, 2009. ACM.

[FM14] Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 1–16, 2014.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[FW10] Zachary P. Fry and Westley Weimer. A human study of fault localization accuracy. In *26th IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[Gio09] Toni Giorgino. Computing and visualizing dynamic time warping alignments in r: The dtw package. *Journal of Statistical Software, Articles*, 31(7):1–24, 2009.

[GRZ18] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 465–480, New York, NY, USA, 2018. Association for Computing Machinery.

[Guo13] Philip J Guo. Online Python tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[HGS$^+$17a] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 89–98, New York, NY, USA, 2017. ACM.

[HGS$^+$17b] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, page 89–98, New York, NY, USA, 2017. Association for Computing Machinery.

[HMBK10] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[JDDS13] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 43–52, New York, NY, USA, 2013. ACM.

[JDJS14] Garvit Juniwal, Alexandre Donzé, Jeff C. Jensen, and Sanjit A. Seshia. Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 24:1–24:10, New York, NY, USA, 2014. ACM.

[JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[JJDS15] Garvit Juniwal, Sakshi Jain, Alexandre Donzé, and Sanjit A. Seshia. Clustering-based active learning for cpsgrader. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 399–403, New York, NY, USA, 2015. ACM.

[JvdBvDH93] Stef Joosten, Klaas van den Berg, and Gerrit van Der Hoeven. Teaching functional programming to first-year students. *J. Funct. Program.*, 3:49–65, 1993.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187 – 195, 1990.

[KLV14] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *Journal of Automated Reasoning*, 52(2):155–190, 2014.

[Koh19] Tobias Kohn. The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 524–530, New York, NY, USA, 2019. ACM.

[KSKG16] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 739–750, New York, NY, USA, 2016. ACM.

[LCSS16] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. A practical framework for type inference error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 781–799, New York, NY, USA, 2016. ACM.

[Lev66] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[LGC06] Benjamin Lerner, Dan Grossman, and Craig Chambers. Seminal: Searching for ml type-error messages. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 63–73, New York, NY, USA, 2006. ACM.

[LLL09] Eelco Lempsink, Sean Leather, and Andres Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM, 2009.

[LNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[LNZ$^+$05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[LR16] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 298–312, New York, NY, USA, 2016. Association for Computing Machinery.

[LZ17] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.*, 1(OOPSLA):92:1–92:30, October 2017.

[MKM11] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. *SIGPLAN Not.*, 46(6):329–342, June 2011.

[MKM13] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

[Mon18] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

[Mül11] D. Müllner. Modern hierarchical, agglomerative clustering algorithms. *ArXiv e-prints*, September 2011.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

[NT03] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 15–26, New York, NY, USA, 2003. ACM.

[PA16] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157 – 173, 2016.

[Pat81] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.

[PCR$^+$03] S. Prabhakararao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and behaviors of end-user programmers with interactive fault localization. In *Human Centric Computing Languages and Environments*, pages 15–22, 2003.

[PHN$^+$15] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969*, 2015.

[PKW14a] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM.

[PKW14b] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM.

[PO11] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[Pop] Bernard James Pope. language-python package for haskell. `https://github.com/bjpop/language-python/tree/v0.5.8`.

[PVG$^+$11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[QMLW13] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Qui14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[RBR05] Joseph R. Ruthruff, Margaret Burnett, and Gregg Rothermel. An empirical study of fault localization for end-user programmers. In *International Conference on Software Engineering*, pages 352–361, 2005.

[RK12] Kelly Rivers and Kenneth R Koedinger. A canonicalizing model for building programming tutors. In *International Conference on Intelligent Tutoring Systems*, pages 591–593. Springer, 2012.

[RK17] Kelly Rivers and Kenneth R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

[RSD$^+$17] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press.

[RSGD18] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *CoRR*, abs/1803.03806, 2018.

[SEC$^+$20] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 16–30, New York, NY, USA, 2020. Association for Computing Machinery.

[SGSL13a] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.

[SGSL13b] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[SJW16] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 228–242, New York, NY, USA, 2016. ACM.

[SL08] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[SLO19] Dowon Song, Myungho Lee, and Hakjoo Oh. Automatic and scalable detection of logical errors in functional programming assignments. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[SM86] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[SPvR$^+$] Gigi Sayfan, Greg Price, Guido van Rossum, Howard Lee, and Jukka Lehtosalo. Mypy: Optional static typing for python. `http://mypy-lang.org`.

[SSC$^+$17] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to blame: Localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, October 2017.

[SSW04] Peter J Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 80–91, New York, NY, USA, 22 September 2004. ACM.

[SY17] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 273–283, New York, NY, USA, 2017. ACM.

[VCJ16] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *PLDI*, 2016.

[VKSB14] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, pages 45–56, New York, NY, USA, 2014. ACM.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

[WPO15] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of IR-based fault localization techniques. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2015.

[WSS18] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 481–495, New York, NY, USA, 2018. Association for Computing Machinery.

[WZCK14] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214. ACM, 2014.

[XM14] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200. IEEE, 2014.

[XQZ$^+$05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.

[YXK$^+$17] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 26(1):4:1–4:30, 2017.

[Zah71] C. T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Comput.*, 20(1):68–86, January 1971.

[Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.

[ZM14] Danfeng Zhang and Andrew C Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL ’14, pages 569–581, New York, NY, USA, 2014. ACM.