2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)

InFix: Automatically Repairing Novice Program Inputs

Madeline Endres
Computer Science and Engineering
University of Michigan
Ann Arbor, MI, USA
endremad@umich.edu

Georgios Sakkas
Computer Science and Engineering
UC San Diego
La Jolla, CA, USA
gsakkas@eng.ucsd.edu

Benjamin Cosman
Computer Science and Engineering
UC San Diego
La Jolla, CA, USA
blcosman@eng.ucsd.edu

Ranjit Jhala
Computer Science and Engineering
UC San Diego
La Jolla, CA, USA
jhala@cs.ucsd.edu

Westley Weimer
Computer Science and Engineering
University of Michigan
Ann Arbor, MI, USA
weimerw@umich.edu

***Abstract*—This paper presents InFix, a technique for automatically fixing erroneous program *inputs* for novice programmers. Unlike comparable existing approaches for automatic debugging and maintenance tasks, InFix repairs input data rather than source code, does not require test cases, and does not require special annotations. Instead, we take advantage of patterns commonly used by novice programmers to automatically create helpful, high quality input repairs. InFix iteratively applies error-message based templates and random mutations based on insights about the debugging behavior of novices. This paper presents an implementation of InFix for Python. We evaluate on 25,995 unique scenarios with input-related errors collected from four years of data from Python Tutor, a free online programming tutoring environment. Our results generalize and scale; compared to previous work, we consider an order of magnitude more unique programs. Overall, InFix is able to repair 94.5% of deterministic input errors. We also present the results of a human study with 97 participants. Surprisingly, this simple approach produces high quality repairs; humans judged the output of InFix to be equally helpful and within 4% of the quality of human-generated repairs.**

***Index Terms*—input repair, novice programs, human study**

I. Introduction

Novice programmers are increasingly turning to online resources beyond the traditional classroom to learn computing [32], [42]. But even as demand soars for such resources, the educational support provided by online tools leaves much room for improvement [8], especially for those students who need the most help [13]. Free tutoring environments, such as Python Tutor [18], seek to close this gap by providing educational support beyond structured course assignments. However, such sites can still suffer from low retention (Section IV), reducing their ability to help students in practice. We hypothesize that one reason for this low retention is the frustration novices face without instructional support; the time spent debugging a single error has been shown to correlate with student frustration [38]. Although student errors extend from simple syntactic mistakes to more involved semantic errors, we observe that a surprisingly-large portion are input-related (e.g., entering `1,2` instead of `1.2`). Therefore, we choose to focus on providing novices with rapid debugging hints and support for input-related errors to decrease debugging time.

Independently, there is limited research into using source-level automatic program repair and fault localization techniques for pedagogical purposes [1], [7], [19], [36], [40], [50]. From reviewing over six million Python Tutor submissions, we observe that 35% of student interactions involve user input as well as source code. We also find that Python Tutor users fixed 6.6% of interpreter errors by only modifying input data. Unfortunately, heavyweight expert-focused automatic program repair tools and their derivatives, such as GenProg [27] and Angelix [33], both focus on source-code transformations and also are unhelpful and confusing for novices [50]. Therefore, novices facing input-related errors are not well-served by extant automatic program repair tools and must instead rely on manual debugging, a time-consuming endeavor.

A technological solution to this problem should find repairs quickly to fit in the student's workflow (Section IV-A, cf. [48]) and be helpful for novices. We observe that novice repairs are generally short, and thus propose to use algorithms that explore the search space of nearby edits. We also note that learners' errors are not uniformly distributed, and many student programs that show defects with the same error message can be fixed using the same mutation. Therefore, a small number of indicative templates can increase search speed. Finally, we note that the structure of student inputs can be unexpectedly complex; programs often contain interdependent input values, making a randomized approach surprisingly effective.

Based on these insights, this paper presents InFix, a randomized search algorithm for automatically repairing program inputs for generic novice programs. InFix uses language-specific error message templates combined with additional randomized mutations to iteratively search for input repairs. InFix does not require test cases or large amounts of training data, instead using the implicit specification of eliminating interpreter errors [20], [40], [49]. InFix also admits a pleasingly parallelizable implementation, improving efficiency.

We evaluate a Python implementation[1] of InFix. Our error message and mutation templates are developed from our observational study characterizing novice Python input patterns and errors. In total, we abstract five error message templates and five additional simple mutations. Each error message template corresponds to a single interpreter message. We evaluate our implementation on 25,995 input-related scenarios arising from 22,282 unique programs from four years of Python Tutor data (Section VI-B). Each scenario contains a program and input that, when run, generate an input-related error. Overall, we find that with just five error message templates and five mutations, InFix repairs 94.5% of input-related scenarios.

As InFix is aimed toward helping novices debug, it is essential that the repairs are helpful and of high quality. Previous work has shown that student-generated source repairs and expert human tutor input hints can be helpful for students by decreasing debugging time and increasing learning [20], [34]. Therefore, we compare InFix's repairs to those developed by the learners themselves. From a human study involving 97 participants, we find that InFix produces high quality repairs. Specifically, participants find InFix's repairs as helpful as human repairs and within 4% of their quality in a statistically significant manner. We conduct this study with both undergraduates and crowdsourced Amazon Mechanical Turk workers.[2] We also find that 80% of our participants often experience input-related errors in their own programming.

In summary, the main contributions of this paper are:

- InFix, a novel template-based search algorithm for repairing erroneous input data for novice programs
- A characterization of common novice input patterns and input repairs for Python programs
- A Python implementation of InFix based on our characterization that fixes 94.5% of 25,995 input-related errors, in one second each on average
- The results of an IRB-approved human study with 97 participants indicating that InFix's repairs are within 4% the quality of, and equally helpful as, student repairs

II. Motivating Examples

In this section, we present two novice Python scenarios with input-related errors. We adapt both examples from actual student programs submitted to Python Tutor. These examples demonstrate the difference between syntactic and semantic input-related errors, providing motivation for InFix's hierarchical use of error message templates and random mutations.

The scenario in Figure 1 exemplifies a syntactic input-related error explainable as a misunderstanding of Python language behavior.[3] The program in this example accepts a float from the user and carries out a calculation based on this input value. Python's `input()` call accepts floats using

Program Code:

```
x = float(input())
print(x * math.e / 2)
```

Erroneous Input	Student Repair	InFix Repair
`26,2`	`29.2`	`4.5`

Python Error Message:

```
ValueError: could not convert string to
    float: '26,2'
```

Fig. 1: Example of a syntactic input-related error.

period-based decimal notation. However, the student entered `26,2` into the interpreter using a comma instead of a period. Because of this, the Python interpreter is unable to cast the input to a float and throws a `ValueError`. Notice that in this simple case, the input that needs to be modified to fix this error is included in the error message itself. Specifically, the error message points out that the program accepts a float, but that `26,2` is not a float. While this error may appear trivial to fix for expert programmers, novices can have surprising difficulty debugging even simple errors [11]. In fact, novices may not even read error messages in some cases [30, Sec. 3], using their existence as a boolean indicator of success or failure.

When designing InFix, therefore, we choose to use error-message templates to take advantage of the copious information some messages include. In particular, we observe that error messages associated with syntactic mistakes contain the richest information for algorithmic repairs. Our implementation includes a template that specifically addresses `ValueErrors` like those in Figure 1. For this scenario, our template is effective: participants find InFix's repair equal in quality and helpfulness to the human-generated repair.

Despite the applicability of error message templates to simple errors, many input-related errors are semantic and program-specific rather than syntactic. Figure 2 exemplifies one such semantic error. The program uses the first two inputs to build a dictionary that is then accessed using the characters in the third input. Unfortunately, the novice includes incorrect values in the third input line, resulting in a `KeyError`. Specifically, the student flips the lines corresponding to the keys and to the values, perhaps indicating a misunderstanding of Python dictionaries. This error is time consuming, taking the student almost three minutes to fix.

This error is significantly more complex than the error from the first example: any repaired input for Figure 2 must satisfy several program-specific constraints. First, `input_b` must have at least as many characters as `input_a`. Second, all characters in `input_c` must also be in `input_a`.

Furthermore, notice that unlike the example in Figure 1, the error message in Figure 2 is not a rich source of debugging hints. However, we observe that repairs similar to the student generated-repair could be created by simple mutations of the

[1]http://web.eecs.umich.edu/~weimerw/data/infix

[2]https://www.mturk.com/worker

[3]All code examples in this paper use Python 3. We note, however, that `input()` has the same behavior as `raw_input()` in Python 2.

Program Code:

```
input_a = input()
input_b = input()
input_c = input()
c_array = []
dictionary = {}
for i in range(len(input_a)):
   dictionary[input_a[i]] = input_b[i]
for j in range(len(input_c)):
   c_array += dictionary[input_c[j]]
print(c_array)
```

Erroneous Input	Student Repair	InFix Repair
abcd	abcd	-Et
*d%#	badc	abcd
#*%*d*%	abcd	-Et

Python Error Message:

```
KeyError: '#'
```

Fig. 2: Example of a semantic input-related error.

original error-generating input. As a result of these observations, we propose using simple mutations to repair erroneous input data when there is not enough information in the error message alone. For these mutations, we consider the input as a whitespace-separated list of tokens. In our evaluation, InFix finds a solution to the scenario in Figure 2 in under 2 seconds, and in our human study, we find InFix's repair to be equivalent in quality and helpfulness to the student's repair.

From our observations, these two examples are indicative of the majority of input-related errors encountered by novices in online tutoring environments. For syntactic errors, such as the error in Figure 1, we observe that novices tend to repair the same error in similar ways and that error messages are rich sources of information. For semantic errors, like the one in Figure 2, fixes are more varied and the error messages are more opaque. These observations motivate our decision to include both error message templates and mutations in InFix; we use error message templates to quickly repair the most common errors and random mutations to address more complex semantic errors. Our observations also inspire our decision to prioritize one category above the other: we only use mutations when there is no applicable error message template.

III. InFix Algorithm

InFix is a randomized search optimization algorithm that iteratively modifies the original erroneous input until either a correct input is found or the maximum number of probes has been exhausted. The key insights behind InFix are that input repairs are often composed from a small number of common mutations and that these mutations are often heavily correlated to specific error messages. Furthermore, for some simple specific errors, student edits are highly predictable.

For the purposes of this paper, since we target generic novice programs, we define a erroneous input as an input that causes a program to raise an interpreter error. In contrast, we define a correct input as one that causes the program to terminate without error. A user-interaction scenario has an input-related error if the programmer runs both erroneous and correct inputs with the same program (not every error-avoiding input need be helpful to novices; we formally investigate repair quality via a human study in Section VI-G). Given two inputs for one program such that the first is erroneous and the second is correct, we refer to the latter as a fix or repair.

In Section III-A we describe the InFix algorithm, in Section III-B we present information on template selection, and in Section III-C we describe a parallel version of InFix.

A. InFix Algorithm Architecture

At a high level, InFix takes six arguments: the program P, the erroneous input I, the original error message M, a template function T, a set of mutations R, and a maximum number of probes N. The template function T's domain consists of a finite set of error messages which uniquely determine a corresponding input mutation. R is a set of additional input mutations. InFix iteratively mutates I until it either finds a repaired input or has tried N mutations. When successful, InFix returns a correct input I', such that $P(I')$ terminates normally without raising any exception. The pseudocode for InFix can be found in Algorithm 1.

Algorithm 1 Main InFix Algorithm

```
Type Definitions:
 1: Type TokSeq : Sequence of Tokens
 2: Type Mutation : TokSeq → TokSeq
 3: Type Message : Error Message
Require:
 4: Program $P$ : Prog
 5: Original erroneous input $I$ : TokSeq
 6: Error message $M$ : Message
 7: Error message template function $T$ : Message → Mutation
 8: Set of mutations $R$ : Mutation Set
 9: Maximum number of probes $N$ : $\mathbb{N}$
10:
11: procedure InFix ($P, I, M, T, R, N$)
12:     $V \leftarrow \emptyset$                    ▷ $V$ is a set of visited inputs
13:     for $n$ in $[1...N]$ do
14:         if $(M \in domain(T)) \wedge (I \notin V)$ then
15:             $mut \leftarrow T(M)$
16:         else
17:             $mut \leftarrow choose(R)$
18:         $V \leftarrow V \cup \{I\}$
19:         $I \leftarrow mut(I)$
20:         $M \leftarrow run(P, I)$
21:         if $M$ is GOOD then break
22: return $minimize(I)$ if $M$ is GOOD else TIMEOUT
```

During each iteration, InFix mutates I using either the error message template function T or a random mutation from R. These two sources of modification are hierarchical: InFix always applies an error message template if possible (line

15). However, if there is no transformer associated with the error message (i.e., M is not in the domain of T: line 14) or if the resulting mutation has already been considered, a random mutation is applied instead. This mutation is chosen unweighted from a set of mutation templates (line 17).

The program P is then invoked on the modified input (line 20). If the run is error free, than the input is minimized and the process terminates. Otherwise, InFix continues iterating until the probe budget has been exhausted. Any minimization approach that finds a correct fix is acceptable. Previous program repair algorithms have used various minimization methods, such as Delta Debugging [27], [51]. In Section VI-D, we discuss the minimization method in our Python instantiation.

B. Template Selection

InFix relies on the selection of templates T and mutations R. In practice, the domain of T is a small set of the most common error messages. Note that in Algorithm 1 there is a single transformation associated with each error message. InFix thus works best when it is possible to identify a highly-effective transformation for the most frequent messages.

However, there are instances where input-related errors that yield the same error message cannot be fixed with the same template (see Section V-B). Therefore, R should contain error-independent transformations associated with common student mutations. For our Python implementation of InFix, the transformations in T and R were developed by characterizing novice Python input-related errors, the results of which are discussed in Section IV. The specific templates and mutations of our implementation are described in Section V.

C. Parallelizing InFix

InFix's structure is pleasingly-parallel. Running multiple searches in parallel can both decrease the time to first repair and also increase the likelihood of finding a repair. In InFix's parallel form, each thread runs the main InFix loop. However, as with similar parallel repair algorithms, there is a tension between possibly repeating work on independent parallel threads and incurring overtime caused by coordination [39].

We propose an approach without online coordination: instead, each thread is seeded with a random mutation of the original erroneous input. This increases the likelihood, but does not guarantee, that threads explore different areas of the search space. Our low-overhead approach is critical to finding repairs online in the timescales associated with novice interactions (see Section IV-A); we empirically evaluate InFix's sensitivity to the number of threads in Section VI-F.

If multiple threads find repairs in the same iteration, InFix selects a repair with the highest statement coverage. We note that parallelization is especially helpful for finding repairs that rely on random mutations rather than on error templates.

IV. Characterization of Novice Python Input Errors

In this section, we present the findings from our *observational study* to characterize the input structure and associated novice interactions of 6,949 scenarios with input-related errors. Each input-related scenario consists of a program, an erroneous input, and a student-generated input repair as defined in Section III. These errors make up approximately 6.6% of all erroneous interactions on Python Tutor [18] from Jan-1-2017 to Dec-31-2017. The results of this study inform our Python adaptation of InFix (Section V). We restrict our observational study to just one year, 2017, to mitigate overfitting: specifically, we will show that the insights and templates derived from 2017 yield an input repair strategy that generalizes across all the years from 2015 to 2018.

In Subsection IV-A, we present a general analysis of all 6,949 scenarios to understand better the size of the programs, the inputs, and the messages most commonly associated with novice input-related errors. In Section IV-B, we present observations from a more in-depth manual examination of 100 randomly-selected scenarios to better understand the structure of erroneous inputs as well as the repairs students made to fix them. These analyses find that input-related errors are varied and that novice input patterns can be, perhaps surprisingly, complex. We also show that some error messages are significantly more common than others and that similar errors are often fixed in similar ways.

A. Erroneous Input-Related Scenarios: Quantitative Analysis

To characterize the 6,949 scenarios in our data set, we consider the average input and program size, the number of `input` calls per program, the time it takes users to generate repairs, and the prevalence of specific error messages. In these scenarios, there are 6,017 unique programs.

Generally, Python Tutor programs with input-related errors are small; the average program length (excluding blank lines and comments) is 17.1 lines. There is a large range, however: some instances have up to 151 lines. The average input size also varies, ranging from 1 to 5,238 characters with an average of 14.2. These inputs can typically be interpreted as white-space separated token lists, though some use custom delimiters. On average, erroneous inputs have 3.1 tokens, although we observe examples with up to 500.

On average, there are only 1.8 textual `input` calls per program (although we observe programs with up to 20 calls). This small number of calls, however, does not necessarily lead to programs with simple input structures. Loops and type constraints often complicate `input` calls (see Section IV-B). Befitting their varying complexities, input-related errors take a wide range of time for Python Tutor users to resolve. The median time to solve input-related errors is 49 seconds, however 34.6% take users over two minutes to solve and 5.5% take over seven minutes.

The most common error message associated with input-related errors is `ValueError`. Out of the 6,949 scenarios, `ValueErrors` are generated in 3,785 (or 54.5%). The next most common error messages are `IndexError` (1290 scenarios) and `NameError` (778 scenarios). We also note that `ValueError` has several variations, differentiated by the trailing error message text. The most common of these are `invalid literal for int() with base`

10 (2,392 scenarios), could not convert string to float (599 scenarios), and not enough/too many values to unpack (632 scenarios). Together, these ValueError variations account for 3,623 scenarios (52.1%).

B. *Erroneous Input-Related Scenarios: Qualitative Analysis*

We now present qualitative analysis of 100 randomly-sampled erroneous input-related scenarios. Generally, we find that inputs consist almost exclusively of string, integer, and float literals. We also note that integers and floats in erroneous inputs are typically small and positive; of the 77 inputs in this sample with numerical literals, only 12 (15%) of them involve a number above 15 and only one contains a negative. This pattern is similar in the student-generated fixes: only 25/97 number-containing fixes have a number over 15. We also note that in our sample, more fixes (97/100) contain numerical values than erroneous inputs do (77/100). Therefore, we conclude that many errors are caused by omitting a numerical literal required by the program's input structure. We further observe that such errors are typically resolved through the insertion of a number into the erroneous input.

Subjectively, we note that many erroneous inputs likely imply a misunderstanding of Python data formatting. These syntactic errors fall into two main groups: erroneous string-to-type conversions and mistakes involving library functions. Altogether, these account for 33/100 errors in our sample.

The first group involves misunderstanding how literals are represented as strings. For example, we observe novices who use commas instead of periods for decimal notation or who include quotes around numbers. Our data set contains one extreme example where the student entered math.pi/6 to a program expecting an integer. These errors typically result in a ValueError, and students typically fix them by correcting the format. While students often preserve the original numerical value in the repair, this is not always the case (see Figure 1). This indicates that the fix's exact numerical value is not always as important as correcting the formatting.

The second group of syntactic errors involves misunderstanding Python's input and split behavior. By default, input reads until the next newline, and split breaks strings on whitespace. Some students, however, may be unclear on both default behaviors. For example, we observe student attempts to use comma-separated lists or to include the data for multiple input calls on the same line.

The remaining errors in our sample are semantic. These errors are diverse, ranging from simple swapped input orderings to complicated indirections. For example, one program which accepts a list of integers raises an error if there is a duplicate in the list. Student input fixes are similarly diverse. However, we note that elements in the erroneous input are often permuted in the fixed input. We also notice that other elements are often slightly modified in the fix (for example, abcb to abdb). We further observe student fixes that insert strings occurring as literals in the program's source code. For example, one program contains a dictionary with the hard-coded key "pollution". However, the student entered the misspelled polution, inducing a KeyError.

Intriguingly, we find that, despite a small number of static input calls in a given program, exhaustively specifying the set of valid inputs can be challenging. While 42/100 scenario programs only have one static input call, in 17 (40.5%), that call is embedded in either a loop or a split, resulting in a complex dynamic input structure.

We also observe that the length of one input portion is often dependent on the value of a different input portion. We consider two indicative cases: dependent list lengths and sentinel values. Figure 3a shows a Python program where the number of times the second input is triggered depends on the value of the first input. We find a version of this value-dependence in 21/100 scenarios. Figure 3b depicts a sentinel loop pattern where exiting the loop requires a specific input value. We observe instantiations of pattern b in 10% of sampled programs. We further note that errors involving these patterns often relate to value interdependencies. For example, many IndexErrors were caused when one input was used as an index into a structure created by another input. Students typically resolve these errors by inserting or deleting tokens.

Finally, we observe that student fixes for input-related Python errors are generative as well as corrective. As Python is interpreted, no more input is accepted once an uncaught error is thrown. Therefore, when a program with multiple input calls aborts after the first call, a fix must both correct this first error and also generate additional input data for any remaining calls. Of the 100 scenarios we examine, 31 fixes generate additional input. We sometimes see novices submit multiple erroneous inputs before achieving an error-free program run, indicating that debugging for novices is an iterative process. An example of this pattern is shown in Figure 3c.

Overall, novice inputs and input-related errors are varied and complex. Input-related errors range from simple syntactic mistakes to more insidious semantic errors that often expose tricky interdependencies between input values and calls. Despite this variety, we note that fixes often contain similar mutations of the original erroneous input. We also find that, for simple syntactic errors that result in a ValueError, the rich error message is highly predictive of the eventual fix.

V. Specializing InFix to Python

To create Python-specific error message templates and mutations for InFix, we exploit patterns discovered in our observational study. We design error message templates to quickly target syntactic errors and select mutations inspired by common debugging patterns for semantic errors. Section V-A discusses the specific error message templates, and Section V-B presents our mutation operators. In total, our InFix implementation contains five error message based templates and five mutations.

A. *Error Message Templates*

In our observational study (see Section IV-A), we discover that just four subtypes of ValueError account for 52.1%

```
num = int(input())
val = [int(x) for x in
    input().split()]
for y in len(num):
    print(val[y])
```

(a) Example where the length of the second input depends on the value of the first: list `val` must be at least as long as `num`. E.g., this program accepts `2\n1 2` and not `2\n1`.

```
while True:
    value =
        int(input("Number?"))
    if value == 10:
      print("Found Number")
      break
```

(b) Example of a sentinel input pattern. To terminate, the input must be exactly equal to 10.

```
name = input("Name?")
age = float(input("Age?"))
num = int(input("Fave num?"))
```

Try	Input	Error
1	`Bob 1,21`	`ValueError`
2	`Bob 9.5 3.4`	`ValueError`
3	`Bob 21 345`	No Error

(c) Example of iterative debugging process. Note: for space, `input` shown is whitespace-separated.

Fig. 3: Common Python input and debugging patterns for novice programmers.

of all input-related errors encountered by novices on Python Tutor. We further find that many of these `ValueErrors` correspond to syntactic errors such as incorrect Python type formatting or misunderstood `input` and `split` behavior, noting that students fix these errors in predictable ways. Additionally, we realize that novice fixes for input-related errors are often necessarily generative. Based on these observations, we implement five error message templates: four that address the most common `ValueError` subtypes and one that addresses `EOFErrors` generated when the program runs out of input. Table I shows our error message templates.

B. Mutation Templates

While our error message templates directly address a significant portion of syntactic errors, they do not cover all errors. In our observational study, we found that the student fixes could often be generated by applying a small set of mutations. From this finding, we propose four mutations: inserting a new token, transforming a space-separated list into one separated by newlines, reordering tokens, and deleting tokens. While many fixes are similar to the original input, we also observed instances where they were seemingly unrelated. As a result, we create a fifth mutation that clears the given input. Table I summarizes the mutations in our InFix implementation.

VI. Evaluation

This section contains the experimental setup and results of our InFix implementation evaluation. Section VI-A outlines our research questions, Section VI-B describes the expanded Python Tutor Data set we used for our evaluation, Section VI-C describes our human study methodology, and the remaining subsections present the results of our investigations into each research question.

A. Research Questions

In our evaluation, we focus on five research questions:

- RQ1: How effective is InFix at repairing Python input-related errors?
- RQ2: Are the assumptions behind InFix, such as the benefits of the hierarchical ordering between error messages and templates, valid?
- RQ3: How sensitive is InFix to available resources?
- RQ4: What are the quality and helpfulness of the repairs produced by InFix, as judged by humans?
- RQ5: How do the perceived helpfulness and quality of these repairs vary with programmer expertise?

B. Benchmark 1: Python Tutor Data Set

Our first evaluation benchmark consists of 25,995 erroneous input-related scenarios collected from four years of Python Tutor data (Jan-1-2015 to Dec-31-2018). Each scenario consists of a Python program, an error-causing input, and a student-generated repair. By year, there are 1,640 scenarios from 2015, 4,683 from 2016, 6,949 from 2017, and 12,723 from 2018. This data is an expansion of the data used in our observational study described in Section IV. To mitigate overfitting, we only used a subset of the data earlier. Across these scenarios, there are 22,282 submissions from 13,968 unique IP addresses. We find that 69.8% of users are single-time users, only ever submitting one input-related error to Python Tutor.

C. Benchmark 2: Repair Quality Human Study

Our second data source is an IRB-approved human study with 97 participants. This data includes quality and helpfulness ratings for InFix- and student-generated repairs for 60 scenarios randomly selected from the Python Tutor data without adaptation. Of the 97 participants, 24 are undergraduate or graduate students at the University of Michigan. The remaining 73 are workers recruited from Amazon Mechanical Turk (MTurk). These participants have varying levels of self-reported Python programming experience.

Each participant was shown an online series of 16 novice Python programs randomly selected from the 60 stimuli corpus.[4]. Each stimulus consists of a Python program, erroneous input, error message, repaired input, and repaired output. There are two versions of each stimulus, one with the historical student-generated repair and one where the repair is generated by InFix. To avoid training effects, a single participant was never shown both the machine and human repair for the same error. For all 16 stimuli, participants were asked to provide a textual description of the cause of the error and to assess the quality and helpfulness of the repair on a Likert scale of 1 to 7. To collect data that best reflected our participants'

[4]All stimuli are available at http://web.eecs.umich.edu/~weimerw/data/infix

Error Message	**Template Fix**
`ValueError: invalid literal for int() with base 10: 'X'`	Replace last instance of `X` with random integer between -1 and 10. Note: novices mostly use small numbers in their fixes.
`ValueError: could not convert string to float: 'X'`	Replace last instance of `X` with random float between -1.00 and 10.00. Note: novices mostly use small numbers in their fixes.
`ValueError: not enough values to unpack`	Append duplicate of last token. Deliminator is either whitespace or extracted from code.
`ValueError: too many values to unpack`	Remove last token from input. Deliminator is either whitespace or extracted from code.
`EOFError: EOF when reading a line`	Append a duplicate of a random token from the original input or append a new random three-character string.

Mutation	**Description**
Insert a token	Inserts token at random location. New token is a short random string, token from original input, or string literal from source.
Split whitespace list	Transforms a line separated by spaces (or other `split` deliminator from source code) and into content separated by newlines.
Swap a token	Swaps a random token with either a short random string, a token from the original input, or a string literal from source code.
Remove a token	Deletes a random token. A token may be an entire line, but we consider whitespace separation when applicable.
Empty the input	Replaces the entire input with an empty sequences. Useful for unhelpful student-provided initial inputs.

TABLE I: Descriptions of error message templates and mutations in InFix's Python implementation.

subjective human experiences we did not further define quality or helpfulness. We also gathered self-reported estimates of both programming experience and Python-specific experience as well as qualitative data pertaining to what factors influence a subjective judgment of repair quality. Study stimuli are very similar to Figure 5, except that participants were only shown one repair per scenario.

The study takes around 45 minutes to complete. A participant's response was only considered valid if it correctly identified the cause of 6 / 16 errors. This high threshold is relevant for trusting MTurk worker ratings. Previous work shows that much of the data submitted on MTurk is of low quality; some users even "collude" to take advantage of the system [24]. MTurk workers were compensated with $4.50 upon successful completion while students could opt to be entered to win one of two $50 Amazon gift cards.

D. RQ1: How Effective is InFix?

We evaluate InFix on 25,995 Python Tutor scenarios with input-related errors. For our initial effectiveness assessment, we set the maximum number of probes-per-thread N to 60 (Section III-A) and the number of threads to five (Section III-C). We perform a sensitivity analysis on these input parameters in Section VI-F. We use a simple brute-force minimization technique (Section III-A): due to the typically-short input length, we find more heavy-weight approaches unnecessary. All experiments were conducted on an Ubuntu 18.04.2 LTS server with a 4.3 GHz Intel i7-7740X quad-core CPU with 32 GB of RAM.

In general, our evaluation demonstrates that InFix is highly effective, able to repair 94.5% of input-related errors. We also find that InFix's repair rate is consistent, achieving similar accuracy for each separate year of data. As InFix's templates were developed from observations of the 2017 data, we believe this consistency indicates our templates generalize. A detailed breakdown of this analysis can be found in Table II.

We also find that InFix is efficient, able to repair the majority of errors in under one second of wall clock time.

TABLE II: Overall InFix evaluation results. All reported results are run with a 60 probe budget and 5 threads. Median and average probe and time costs (wall clock) are shown.

	Input-Error Scenarios			Probes		Time (sec)	
Year	Total	Repaired	%	Med.	Avg.	Med.	Avg.
2015	1,640	1,582	96.5%	1	2.98	0.87	1.12
2016	4,683	4,440	94.8%	2	3.23	0.88	1.16
2017	6,949	6,590	94.8%	2	3.47	0.90	1.23
2018	12,723	11,947	93.9%	2	3.70	0.88	1.28
Total	25,995	24,559	**94.5%**	2	3.50	**0.88**	**1.23**

This is important because InFix is intended to provide real-time debugging hints and repairs to novices.

InFix is both highly effective and efficient, repairing 94.5% of 25,995 input-related scenarios in a median of 0.88 wall clock seconds (vs. 49 median seconds for novices).

E. RQ2: Validating InFix's Design Assumptions

Beyond assessing the overall effectiveness of InFix, we also perform an experiment to validate our design assumptions that both error message templates and randomized mutations are helpful and that error message templates should take precedence. To do so, we implement three variants of InFix: one with only error message templates, one with only mutation templates, and one with both that uses random selection instead of a hierarchical prioritization (see Section III-A).

We compare the performance of these variations on the 1,640 2015 Python Tutor scenarios using five parallel threads and 60 maximum probes-per-thread. We find that InFix outperforms all three variations in repair rate, average number of probes, or both, indicating that the error message templates, mutations, and their associated hierarchy all contribute to InFix's high performance. In particular, the error-message-only and mutation-only implementations have markedly lower repair rates than InFix. Interestingly, we observe that the 45.2% of scenarios that the error-message-only version repairs is similar to the 52.1% of errors we templated (see Section V), indicating that our abstracted error message templates are highly effective. We also note that while the non-hierarchical

TABLE III: Experimental results for validation of InFix's design assumptions. Tested on the 1640 scenarios from 2015.

Algorithm Variation	Number of Inputs Fixed	Percent Fixed	Average Probes To Solve
Error Messages Only	741	45.2%	2.70
Mutations Only	1048	64.5%	10.50
Non-Hierarchical	1561	95.2%	4.10
InFix (complete)	1582	**96.5%**	**2.98**

TABLE IV: InFix's sensitivity to the maximum number of probes and the number of threads. Each box contains the percentage of the programs solved when InFix is run with the specified parameters.

Maximum Number of Probes	Number of Threads: 1	2	3	4	5
1	30.8%	36.4%	39.9%	42.6%	44.6%
5	64.1%	72.7%	77.3%	80.3%	82.6%
10	73.6%	81.0%	84.5%	86.7%	88.4%
20	80.5%	86.1%	88.8%	90.6%	91.7%
30	83.1%	88.2%	90.5%	92.0%	93.0%
60	86.7%	91.0%	92.7%	93.8%	94.5%
500	92.5%	94.5%	95.3%	95.8%	96.1%

version's performance is only slightly lower than InFix's, the average number of probes is 27% greater validating our assumption that the hierarchy between error message templates and mutations leads to increased efficiency. A detailed breakdown of our results is given in Table III.

Error message templates, mutations, and the hierarchical mutation structure are critical for InFix's high performance.

F. RQ3: InFix Parameter Sensitivity

To understand InFix's parameter sensitivity, we evaluate InFix with different probe budgets (1–500) and parallel threads (1–5). We choose to focus on sensitivity with respect to more constrained resources because, unlike traditional automatic program repair, we target real-time repairs for low-budget tutoring sites. We do, however, include a larger probe budget to compare against previous work. For probe budgets $N \leq 60$, we evaluate on all 25,995 scenarios. For 500 probes, we evaluate only 4,683 scenarios from the 2016 Python Tutor data. We find that InFix's repair rate is influenced by the values of these two input parameters; as the probe budget and threads increase, InFix's repair rate also increases. Numerical results from our sensitivity analysis are shown in Table IV.

We emphasize two of our findings. First, note that even with a single probe, InFix repairs a large number of input scenarios. We observe that most of these correspond to instances where the initial error message is templated. This demonstrates that InFix can still be effective even with highly constrained resources. Second, even when the probe budget doubles from 30 to 60, the repair rate with a consistent number of threads increases by at most 4.1%. Between 60 and 500 probes, an 8x resource increase, this pattern is even more pronounced, with a maximum repair-rate increase of 6.3%. This indicates that InFix is largely insensitive to resource constraints on the order of those bounds established by previous work (e.g., up to 307 probes reported for three algorithms on the similarly-sized IntroClass student program repair benchmark [26, Fig. 5]).

TABLE V: Quality and Helpfulness from 1,544 human study ratings (1–7 Likert scale). InFix's patches are 4% lower quality than human-written patches in a statistically-significant manner ($p = 0.047$); helpfulness is not statistically distinguishable.

	Rated Patch Quality			Rated Patch Helpfulness		
Raters	Human	InFix	p-value	Human	InFix	p-value
MTurk	4.7	4.5	0.042	4.7	4.6	0.086
University	4.6	4.5	0.360	4.5	4.8	0.110
All Raters	4.7	4.5	**0.047**	4.7	4.6	0.270

InFix is insensitive to expected resource parameters and is usable even for non-parallel architectures and tight resource budgets. InFix also repairs a non-trivial amount of input-related errors in a single iteration.

G. RQ4: What is the Quality of InFix's Repairs?

As human-generated inputs and code repairs have been shown to be useful hints for novices [20], [34], we objectively and subjectively investigate how InFix's repairs compare to historical repairs made by the Python Tutor users themselves.

Our objective evaluation compares the statement coverage of InFix's repairs to the coverage of student repairs. We choose coverage because it is a well-understood and commonly-used metric for software engineering quality assurance [3], [31].

From analysis on the 2018 Python tutor data, we find that the median coverage of InFix' repairs (83.3%) is 90.2% the median coverage of student repairs (90.3%). InFix's coverage is high, comparable to that achieved by PEX [45], an automated test generation tool evaluated in an educational setting [46], and greater than that of tools such as KATCH that focus on coverage for expert-written patches [31, Tab. 1].

In our second evaluation, we asked humans for subjective assessments of repair quality (see Section VI-C). We collected 1,544 helpfulness and quality scores for machine and student input repairs on a Likert scale between 1 (low) and 7 (high). Details of our results are in Table V, including subgroup breakdowns for university students and MTurk workers. Overall, InFix's repairs were as helpful as student generated repairs: we found no statistically-significant difference between the helpfulness of student- and machine-generated repairs using the two-tailed Mann-Whitney U test. We did, however, find a statistically-significant difference for repair quality ($p = 0.047$): the quality of human repairs is 4% higher than InFix repairs. This subjective 96% quality assessment is very high compared to previous investigations of automated repairs.[5]

While we deliberately do not define quality and helpfulness to avoid biasing responses, we did ask our participants "what factors cause a repair to be of high quality". Generally, subjects

[5]For example, while not directly comparable, humans found PAR's patches 75.5% as acceptable as human patches and GenProg's 51.4% as acceptable [23, Tab. VII]. Similarly, Long and Rinard report 18 of Prophet's 39 patches to be correct [29, Fig. 10] in a manual human assessment, with other algorithms such as GenProg and Kali performing worse.

Program Code:

```
ticket_num = int(input("How many?: "))
cost = float(input("How much each?: "))
total = ticket_num * cost
print("Your cost is", total, ".")
```

	Erroneous Input	Student Repair	InFix Repair
Input	3 $1.50	3 1.50	3 8.10
Output	Error	Your cost is 4.5 .	Your cost is 24.2999997 .

Python Error Message:

```
ValueError: could not convert string to
    float: '$1.50'
```

Fig. 4: Example where the *machine* repair is of higher quality than the student repair by 1.4 ($p = 0.028$).

Program Code:

```
1   day = int(input("enter a day :"))
2   month = int(input("enter a month :"))
3   year = int(input("enter a year :"))
4
5   def leap(year):
6      pass # Removed for space considerations
7
8   def checkDay(day, month, year):
9      if day == 29 and month == 2 and leap():
10        return day
11     return False
12
13  print(checkDay(day, month, year))
```

Erroneous Input	Student Repair	InFix Repair
29 2 2016	12 2 2000	6 10 2

Python Error Message:

```
TypeError: leap() missing 1 required
    positional argument: 'year'
```

Fig. 5: Example where the *human* repair is of higher quality than the student repair ($p = 0.019$).

indicate that high-quality repairs are those that help with fault localization. For example, one participant stated a repair is of higher quality "the quicker it helped [her] solve the bugged input". Similarly, a second participant wrote that high-quality repairs "provide a valuable clue . . . [because] when you follow the code and use [the] new input, the error is easier to spot". One participant articulated that high-quality repairs should "exercise as much code as possible instead of giving values that short-circuit checks or skip faulty code", supporting our use of code coverage as a proxy for repair quality.

To further tease apart this nuanced notion of repair quality, we consider two case studies with statistically-significant differences between human and machine repairs. Figure 4 shows an example where the machine repair is rated better than the human repair ($p = 0.028$). This program asks the user to enter a monetary amount. The erroneous input contains a simple syntactic error: the novice includes a dollar sign with the float. The human repair simply removes the dollar sign. InFix, however, suggests a different float. Perhaps unexpectedly, participants find the machine repair of higher quality than the human repair. We hypothesize this is because only the machine repair's output reveals floating point precision formatting behavior undesirable for monetary notation.

In contrast, Figure 5 depicts an example where participants thought the human repair was better than the machine repair ($p = 0.019$). In this program, the input-related error is primarily caused by a defect in the code: on line 9, the programmer incorrectly calls `leap()`. The student input fix includes a different year, avoiding the defect due to short-circuit evaluation. InFix, however, generates `2` for the year. We believe the fact that `2` makes no contextual sense as a modern year to be the reason for its lower perceived quality. In fact, one participant singled out this repair as particularly poor, noting that "a valid date . . . would give a better example than a wrong (logically) year value of 2".

These two examples demonstrate that the stochastic elements of InFix can benefit repair quality, revealing edge cases that would otherwise be missed. However, they also show a limitation of template-based repair: we deliberately used small numerical values in our repairs based off our observations in Section IV. However, this is unhelpful when the student program involves numerical inputs with other contextual constraints (Figure 5).

InFix's repairs are of high quality, attaining 90.2% of the statement coverage of student repairs. More importantly, 97 study participants found InFix's repairs to be equally helpful as, and to have 96% of the quality of, human repairs.

H. RQ5: The Effect of Programmer Expertise

As InFix is designed to help novices, we are interested in the effect of programmer expertise on repair helpfulness. We analyzed the helpfulness scores of experience-based subpopulations. In our human study (see Section VI-C), participants were asked to self-report their Python experience as either minimal (less than a semester), moderate (1–2 semesters), or expert (3+ semesters). Of the 96 respondents, 31 are minimal, 49 are moderate, and 16 are experts. While we note we use a coarse definition of programmer expertise (see Siegmund et al. [43] for a detailed discussion), we claim that students with three or more semesters programming experience are relative experts compared to those who have just started learning.

Initially, we observed that our relative experts rated InFix repairs more helpful than novices (4.0 vs. 5.2 out of 7). However, since the stimuli for our study were randomly sampled from the Python Tutor data set, they vary in difficulty: some programs contain Python language features that participants with minimal experience may not have encountered. We

		Participant Experience Level		
	#	Minimal	Moderate	Expert
Easiest Stimuli	14	4.7	4.9	5.1
Hardest Stimuli	11	3.6	4.8	5.1
All Stimuli	60	4.0	4.8	5.2

TABLE VI: Helpfulness ratings of InFix's repairs depending on experience. Scores are on a scale from 1 to 7.

hypothesize that these hard programs are confusing for our most novice participants, leading to lower helpfulness ratings.

We thus analyzed the helpfulness of InFix's repairs for the easiest 14 and hardest 11 programs as determined by three expert annotators (Fleiss $\kappa = 0.71$). We find that participants with the least experience give repairs for easy programs higher helpfulness ratings than they give repairs for the hardest programs (4.7 vs. 3.6). Participants with the most experience, however, rate machine repairs for both easy and hard programs as equally helpful. For the easiest programs, there is no statistically-significant difference in the helpfulness scores between novice and relative expert participants, indicating that InFix is helpful for novice programmers with varying experience levels. Our results are detailed in Table VI.

After controlling for program difficulty, InFix's repairs are rated equally helpful by developers with varying Python experience, including novices ("less than a semester").

I. Evaluation Summary

InFix is highly effective and efficient enough to help students debug in real time. InFix is relatively insensitive to resource-based parameters, indicating that input-related repair can be cost effective to deploy under constraint. Beyond its high 94.5% repair rate and sub-second efficiency, we find that InFix produces very high quality repairs (96%) that are helpful for novices and experts alike.

VII. Threats to Validity

Although our experiments indicate that InFix is effective and efficient, our results and subjective quality data may not generalize. We also consider that novices may feel they rarely encounter input-related errors, making repairs unnecessary.

We first recognize that while InFix is highly effective for Python, our results may not generalize to other languages. We find it likely that InFix's success depends on the expressiveness of the language's error messages. We deliberately implement InFix for a language widely used by novices. However, investigating cross-language effectiveness remains for future work.

To mitigate the possibility of fraudulent MTurk data, we require workers to correctly identify the cause of at least 6 / 16 stimuli errors for payment. We only considered responses meeting that threshold, as assessed through manual analysis, in our evaluation. Filtering for data set inclusion based on response quality is a best practice for studies involving crowdsourced participants [12]. While 186 workers requested payment on MTurk, only 73 (39%) met the validity threshold for inclusion. This MTurk retention rate is similar to that reported by other crowdsourced studies involving debugging [14].

Finally, to mitigate threats involving problem significance, we asked our study participants how strongly they agree with the phrase: "I often encounter bugs where the input data is part of the problem". Participants report commonly encountering input-related errors: 75 / 94 responses (80%) agree or strongly agree, while only three participants strongly disagree. This result indicates that input-related errors are a common and memorable challenge faced by novices.

VIII. Related Work

A. Pedagogical Automatic Program Repair

Previous pedagogically-motivated automatic program repair and fault localization work focuses on large course assignments (e.g. MOOCS) [1], [7], [26], [36], [50] rather than support for non-traditional students. For example, Ahmed et al. build statistical models to help repair submissions for 14 different problem sets, using between 400 and 9,000 submissions per problem for training [1]. However, as we focus on input-related errors for generic student programs, InFix must operate without a large corpus of fixes for the same program.

Yi et al. study if state-of-the-art program repair tools can feasibly help students repair source-level errors [50]. They find that expert-focused tools and their derivatives are unhelpful, though they are potentially useful for course graders. However, they did not investigate input-related repairs.

B. Automatic Input Rectification, Sanitization or Fuzzing

Limited work has been done on automatically repairing input data [2], [28], [35]. Extant work focuses on improving security for industrial programs. In the most related work, Long et al. use provided tests to learn what non-malicious inputs look like for a program [28]. They then automatically correct "atypical" inputs to fit the learned pattern. For general novice programs, however, the input format is rarely specified and there are rarely test cases. To the best of our knowledge, there is no prior work on automatically repairing novice input errors or investigating their repair quality.

Given a model for functions such as `input`, `split` and `int`, fuzz testing (e.g., [15], [16], [41]) could be applied to the task of generating non-erroneous inputs. However, test input generators often struggle with real-time answers to semantic or dependent input paths, such as the dictionary key-value scenario in Figure 2. While a few algorithms have efficient web deployments, such as Pex [46], approaches that handle complex input constraints, such as EXE, generally require minutes [10] rather than seconds. In addition, while there are some evaluations of fuzzing in pedagogical contexts (e.g., as a game [47]), we are unaware of any work evaluating fuzzing quality for novice input repairs. We view a more thorough evaluation of fuzz testing in this context as future work.

C. Intelligent tutoring systems

There exists a large body of work investigating and evaluating intelligent tutoring systems for learning programming [4], [9], [20], [21], [25], [37], [44]. These systems target a wide range of programming languages and experience levels. Many of these systems provide data-driven source code fixes to serve as general hints for learning [20], [25], [37]. For example, Hartmann et al. use crowdsourcing to provide selected solutions to error messages [20]. Others, such as Singh et al., take advantage of reference implementation to provide more specific feedback [44]. Others also use static analysis or constraint solving to provide state based hints [18], [21].

In approaches similar to our own, Lazar et al. provide hints using common student edits [25] and Berges et al. characterize common novice error messages [6]. Both, however, focus solely on source-level errors while we focus on input-related errors. To the best of our knowledge, we are the first paper to either classify or repair common novice input-related errors.

D. *Input Grammar Generation*

While InFix uses randomization to generate input repairs, it only generates a single fix. We hypothesize that automatically synthesizing input grammars for student programs could result in richer information for providing hints. Synthesizing generic input grammars remains a challenging task. However, there has been some recent work in this area [5], [17], [22]. For example, Bastani et al. develop an algorithm for synthesizing context-free input grammars [5]. Unfortunately, our characterization of novice input structures in Section IV found that many are actually context-sensitive. Automatically generating context-sensitive input grammars remains an area for future work.

IX. Conclusion

This paper presents InFix, a randomized template-based approach for automatically fixing erroneous program inputs for novice programmers. InFix repairs input data rather than source code, requires no test cases, and requires no special annotations. We take advantage of novice inputs patterns that we characterized in an observational study to automatically create helpful, high quality input repairs. InFix iteratively applies prioritized error-based templates and random mutations.

We evaluate on 25,995 unique input-related scenarios from over four years of data. Our results generalize and scale; compared to previous work, we consider an order of magnitude more unique programs. Overall, InFix repaired 94.5% of input errors. The majority were repaired in under a second, facilitating real-time repairs. We also present the results of a human study with 97 participants. InFix produces high quality repairs: humans judged the output of InFix to be equally helpful and within 4% of the quality of human-generated repairs. Insensitive to expected resource parameters, InFix is usable even for environments with tight resource budgets.

Acknowledgments

We acknowledge the partial support of the NSF (CCF 1908633, CCF 1763674) and the Air Force (FA8750-19-2-0006, FA8750-19-1-0501). We also thank the other members of the WRG Lab for their willingness to answer questions. In particular, we thank Kevin Angstadt for sharing his code from a previous human study website and Fee Christoph who provided additional annotations for our evaluation. Finally, we extend our gratitude to Philip Guo who was generous enough to share the Python Tutor data.

References

[1] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani. Compilation error repair: for the student programs, from the student programs. In *International Conference on Software Engineering*, pages 78–87, 2018.

[2] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis*, pages 225–236, 2014.

[3] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[4] T. Barnes and J. C. Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In *Intelligent Tutoring Systems*, volume 5091, pages 373–382, 2008.

[5] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *Programming Language Design and Implementation*, pages 95–110, 2017.

[6] M. Berges, M. Striewe, P. Shah, M. Goedicke, and P. Hubwieser. Towards deriving programming competencies from student errors. In *International Conference on Learning and Teaching in Computing*, pages 19–23, 2016.

[7] G. Birch, B. Fischer, and M. Poppleton. Using fast model-based fault localisation to aid students in self-guided program repair and to improve assessment. In *Innovation and Technology in Computer Science Education*, pages 168–173, 2016.

[8] C. J. Bonk, M. M. Lee, X. Kou, S. Xu, and F. Sheu. Understanding the self-directed online learning preferences, goals, achievements, and challenges of MIT opencourseware subscribers. *Educational Technology & Society*, 18(2):349–365, 2015.

[9] C. J. Butz, S. Hua, and R. B. Maguire. A web-based bayesian intelligent tutoring system for computer programming. *Web Intelligence and Agent Systems*, 4(1):77–97, 2006.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.

[11] P. Denny, A. Luxton-Reilly, and D. Carpenter. Enhancing syntax error messages appears ineffectual. In *Innovation and Technology in Computer Science Education*, pages 273–278, 2014.

[12] J. S. Downs, M. B. Holbrook, S. Sheng, and L. F. Cranor. Are your participants gaming the system?: screening Mechanical Turk workers. In *Human Factors in Computing Systems*, pages 2399–2402, 2010.

[13] S. Dynarski. Online courses are harming the students who need the most help. In *https://www.nytimes.com/2018/01/19/business/online-courses-are-harming-the-students-who-need-the-most-help.html*, 2018.

[14] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10. IEEE Computer Society, 2010.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[17] R. Gopinath, B. Mathis, M. Höschele, A. Kampmann, and A. Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *CoRR*, abs/1810.08289, 2018.

[18] P. J. Guo. Online Python Tutor: embeddable web-based program visualization for CS education. In *Symposium on Computer Science Education*, pages 579–584, 2013.

[19] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. DeepFix: Fixing common C language errors by deep learning. In *Conference on Artificial Intelligence*, pages 1345–1351, 2017.

[20] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Human Factors in Computing Systems*, pages 1019–1028, 2010.

[21] J. Holland, A. Mitrovic, and B. Martin. J-LATTE: a constraint-based tutor for Java. In *International Conference on Computers in Education*, pages 1–5, 2009.

[22] M. Höschele and A. Zeller. Mining input grammars with AUTOGRAM. In *International Conference on Software Engineering*, pages 31–34, 2017.

[23] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, pages 802–811, 2013.

[24] A. Kittur, J. V. Nickerson, M. S. Bernstein, E. Gerber, A. D. Shaw, J. Zimmerman, M. Lease, and J. J. Horton. The future of crowd work. In *Computer Supported Cooperative Work*, pages 1301–1318, 2013.

[25] T. Lazar and I. Bratko. Data-driven program synthesis for hint generation in programming tutors. In *Intelligent Tutoring Systems*, volume 8474 of *Lecture Notes in Computer Science*, pages 306–311, 2014.

[26] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256, 2015.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[28] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. C. Rinard. Automatic input rectification. In *International Conference on Software Engineering*, pages 80–90, 2012.

[29] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Principles of Programming Languages*, pages 298–312, 2016.

[30] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Symposium on New Ideas in Programming and Reflections on Software, Onward!*, pages 3–18, 2011.

[31] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *Foundations of Software Engineering*, pages 235–245, 2013.

[32] A. McCarthy. Most popular MITx MOOC reaches 1.2 million enrollments. In *http://news.mit.edu/2018/first-mitx-mooc-reaches-enrollment-milestone-0830*, 2018.

[33] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, pages 691–701, 2016.

[34] D. C. Merrill, B. J. Reiser, S. K. Merrill, and S. Landes. Tutoring: Guided learning by doing. *Cognition and Instruction*, 13(3):315–372, 1995.

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

[36] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya. Automatic grading and feedback using program repair for introductory programming courses. In *Innovation and Technology in Computer Science Education*, pages 92–97, 2017.

[37] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *I. J. Artificial Intelligence in Education*, 27(1):37–64, 2017.

[38] M. M. T. Rodrigo and R. S. J. de Baker. Coarse-grained detection of student frustration in an introductory programming course. In *International Workshop on Computing Education Research*, pages 75–80, 2009.

[39] E. M. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems*, pages 317–328, 2013.

[40] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM on Programming Languages*, 1(OOPSLA):60:1–60:27, 2017.

[41] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 263–272, 2005.

[42] D. Shah. A product at every price: A review of MOOC stats and trends in 2017. In *https://www.edsurge.com/news/2018-01-22-a-product-at-every-price-a-review-of-mooc-stats-and-trends-in-2017*, 2018.

[43] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[44] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Programming Language Design and Implementation*, pages 15–26, 2013.

[45] N. Tillmann and J. de Halleux. Pex — white box test generation for .NET. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153, 2008.

[46] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Pex4Fun: A web-based environment for educational gaming via automated test generation. In *International Conference on Automated Software*, pages 730–733, 2013.

[47] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Constructing coding duels in Pex4Fun and code hunt. In *International Symposium on Software Testing and Analysis*, pages 445–448, 2014.

[48] M. Wall. How long will you wait for a shopping website to load. In *https://www.bbc.com/news/business-37100091*, 2016.

[49] C. Watson, F. W. B. Li, and J. L. Godwin. BlueFix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Advances in Web-Based Learning*, volume 7558 of *Lecture Notes in Computer Science*, pages 228–239, 2012.

[50] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Foundations of Software Engineering*, pages 740–751, 2017.

[51] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.