Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models*

Tung Phung[1]
MPI-SWS
mphung@mpi-sws.org

José Cambronero[2]
Microsoft
jcambronero@microsoft.com

Sumit Gulwani[2]
Microsoft
sumitg@microsoft.com

Tobias Kohn[2]
TU Wien
tobias.kohn@tuwien.ac.at

Rupak Majumdar[2]
MPI-SWS
rupak@mpi-sws.org

Adish Singla[2]
MPI-SWS
adishs@mpi-sws.org

Gustavo Soares[2]
Microsoft
gsoares@microsoft.com

ABSTRACT

Large language models (LLMs), such as Codex, hold great promise in enhancing programming education by automatically generating feedback for students. We investigate using LLMs to generate feedback for fixing syntax errors in Python programs, a key scenario in introductory programming. More concretely, given a student's buggy program, our goal is to generate feedback comprising a fixed program along with a natural language explanation describing the errors/fixes, inspired by how a human tutor would give feedback. While using LLMs is promising, the critical challenge is to ensure high precision in the generated feedback, which is imperative before deploying such technology in classrooms. The main research question we study is: *Can we develop LLMs-based feedback generation techniques with a tunable precision parameter, giving educators quality control over the feedback that students receive?* To this end, we introduce PyFiXV, our technique to generate high-precision feedback powered by Codex. The key idea behind PyFiXV is to use a novel run-time validation mechanism to decide whether the generated feedback is suitable for sharing with the student; notably, this validation mechanism also provides a precision knob to educators. We perform an extensive evaluation using two real-world datasets of Python programs with syntax errors and show the efficacy of PyFiXV in generating high-precision feedback.

Keywords

Programming education, Python programs, syntax errors, feedback generation, large language models

*[1]: Corresponding author.
[2]: Listed in alphabetical order.

1. INTRODUCTION

Large language models (LLMs) trained on text and code have the potential to power next-generation AI-driven educational technologies and drastically improve the landscape of computing education. One of such popular LLMs is OpenAI's Codex [1], a variant of the 175 billion parameter model GPT-3 [2], trained by fine-tuning GPT-3 on code from over 50 million GitHub repositories. A recent study ranked Codex in the top quartile w.r.t. students in a large introductory programming course [3]. Subsequently, recent works have shown promising results in using Codex on various programming education scenarios, including generating new programming assignments [4], providing code explanations [5], and enhancing programming-error-messages [6].

We investigate the use of LLMs to generate feedback for programming syntax errors, a key scenario in introductory programming education. Even though such errors typically require small fixes and are easily explainable by human tutors, they can pose a major hurdle in learning for novice students [7]. Moreover, the programming-error-messages provided by the default programming environment are often cryptic and unable to provide explicable feedback to students [8–10]. Ideally, a human tutor would help a novice student by providing detailed feedback describing the errors and required fixes to the buggy program; however, it is extremely tedious/challenging to provide feedback at scale given the growing enrollments in introductory programming courses [11, 12]. To this end, our goal is to automate the feedback generation process using LLMs-based techniques.

More concretely, given a student's buggy program, we want to generate feedback comprising a fixed program and a natural language explanation describing the errors/fixes, inspired by how a human tutor would give feedback. While models like Codex, trained on both text and code, are naturally suitable for this, the critical challenge is to ensure high precision in the generated feedback. High precision is imperative in building educators' trust before deploying such an AI-driven technology in classrooms. A recent work investigated enhancing the default programming-error-messages using Codex [6]; one of the takeaways, quoted from their paper, is *"The key implications of this work are that program-*

```
print "x"
a = input("xxx")
print  "Xx x"a
b = input("xxx xxx?")
print "xxxxxx" a + b
```

(a) Student's buggy program

```
print "x"
a = input("xxx")
print  "Xx x"+a
b = input("xxx xxx?")
print "xxxxxx" +a + b
```

(b) Generated fixed program

The student forgets to concatenate the string literal with the variable. We can fix the error by adding a plus sign between the string literal and the variable.

(c) Generated explanation

(d) Validation

Figure 1: Illustrative example showcasing PyFiXV on a buggy Python 2 program from TigerJython [9]. **(a)** Buggy program (with string literals anonymized as sequences of 'x' in our dataset). **(b, c)** Fixed program and explanation generated by PyFiXV. **(d)** PyFiXV's validation mechanism *successfully accepted* the generated feedback as suitable for sharing.

```
for i in range(int(input()):
    l1=[]
    for i in range(int(input())):
        if i==0:
            if i==0:
                l1.appned(n)
            elif(i==(n-1)):
                l1.append(1)
            else:
                l1.append(i)
    print(l1)
```

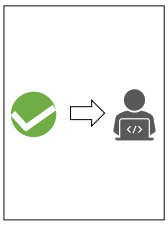
(a) Student's buggy program

```
for i in range(int(input())):
    l1=[]
    for i in range(int(input())):
        if i==0:
            if i==0:
                l1.append(n)
            elif(i==(n-1)):
                l1.append(1)
            else:
                l1.append(i)
    print(l1)
```

(b) Generated fixed program

In line 1, a closing parenthesis is missing when calling function 'range'. We should add a closing parenthesis at the end of line 1, before the colon, to fix this error. In line 6, the function 'append' is misspelled. We should replace 'appned' with 'append' to fix this error.

(c) Generated explanation



(d) Validation

Figure 2: Similar to Figure 1, this example showcases PyFiXV on a buggy Python 3 program from Codeforces [13]. While the misspelling of 'append' as 'appned' is not a syntax error (raising an AttributeError at runtime instead), PyFiXV typically fixes such mistakes as well, showing the generalization value in LLMs-powered technique.

```
import math

def binary(num):
    x=int(math.log2(num))+1
    for i in range(x):
        num=(num ^^(1<<i))
        print(num)
for i in range(int,input()):
    n=int(input())
    binary(n)
```

(a) Student's buggy program

```
import math

def binary(num):
    x=int(math.log2(num))+1
    for i in range(x):
        num=(num ^ (1<<i))
        print(num)
for i in range(int(input())):
    n=int(input())
    binary(n)
```

(b) Generated fixed program

In line 6, the operator '^^' is not a valid operator in Python 3. We should replace it with '^' to fix this error. In line 8, a closing parenthesis is missing when calling function 'range'. We should add a closing parenthesis at the end of line 8, before the colon, to fix this error.

(c) Generated explanation

(d) Validation

Figure 3: Similar to Figure 2, this example showcases PyFiXV on a buggy Python 3 program from Codeforces [13]. PyFiXV's validation mechanism *successfully rejected* the generated feedback (we marked text in **(c)** to highlight issues with explanation).

ming error message explanations and suggested fixes generated by LLMs are not yet ready for production use in introductory programming classes...". Our initial experiments (Section 4) also highlight issues in generating high-precision feedback. To this end, the main research question is:

Can we develop LLMs-based feedback generation techniques with a tunable precision parameter, giving educators quality control over the feedback that students receive?

1.1 Our Approach and Contributions

In this paper, we develop PyFiXV, our technique to generate high-precision feedback powered by Codex. Given a student's buggy program as input, PyFiXV decomposes the overall process into (i) feedback generation (i.e., a fixed program and a natural language explanation for errors/fixes); and (ii) feedback validation (i.e., deciding whether the generated feedback is suitable for sharing with the student). One of the key ideas in PyFiXV is to use a run-time feedback validation mechanism that decides whether the generated feedback is of good quality. This validation mechanism uses Codex as a *simulated student model* – the intuition is that a good quality explanation, when provided as Codex's prompt instruction, should increase Codex's success in converting the buggy program to the fixed program. Notably, this validation also provides a tuneable precision knob to educators to control the precision and coverage trade-off. The illustrative examples in Figures 1, 2, and 3 showcase PyFiXV on three different student's buggy programs. Our main contributions are:

(I) We formalize the problem of generating high-precision feedback for programming syntax errors using LLMs, where feedback comprises a fixed program and a natural language explanation. (Section 2)

(II) We develop a novel technique, PyFiXV, that generates feedback using Codex and has a run-time feedback validation mechanism to decide whether the generated feedback is suitable for sharing. (Section 3)

(III) We perform extensive evaluations using two real-world datasets of Python programs with syntax errors and showcase the efficacy of PyFiXV. We publicly release the implementation of PyFiXV. (Section 4)[1]

[1]Github: `https://github.com/machine-teaching-group/edm2023_PyFiXV`

1.2 Related Work

Feedback generation for programming errors. There has been extensive work on feedback generation for syntactic/semantic programming errors [14–18]; however, these works have focused on fixing/repairing buggy programs without providing explanations. The work in [11] proposed a technique to generate explanations; however, it requires prespecified rules that map errors to explanations. Another line of work, complementary to ours, has explored crowdsourcing approaches to obtain explanations provided by other students/tutors [19, 20]. There has also been extensive work on improving the programming-error-messages by designing customized environments [9, 10]. As discussed earlier, a recent study used Codex to enhance these error messages [6]; however, our work is different as we focus on generating high-precision feedback with a tuneable precision knob.

Validation of generated content. In recent work, [21] developed a technique to validate LLMs' output in the context of program synthesis. While similar in spirit, their validation mechanism is different and operates by asking LLMs to generate predicates for testing the synthesized programs. Another possible approach is to use back-translation models to validate the generated content [22, 23]; however, such a back-translation model (that generates buggy programs from explanations) is not readily available for our setting. Another approach, complementary to ours, is to use human-in-the-loop for validating low confidence outputs [24].

2. PROBLEM SETUP

Next, we introduce definitions and formalize our objective.

2.1 Preliminaries

Student's buggy program. Consider a student working on a programming assignment who has written a buggy program with syntax errors, such as shown in Figures 1a, 2a, and 3a. Formally, these syntax errors are defined by the underlying parser of the programming language [14]; we will use the Python programming language in our evaluation. Henceforth, we denote such a buggy program as $\mathcal{P}_\text{b}$, which is provided as an input to feedback generation techniques.

Feedback style. Given $\mathcal{P}_\text{b}$, we seek to generate feedback comprising a fixed program along with a natural language explanation describing the errors and fixes. This feedback style is inspired by how a human tutor would give feedback to novice students in introductory programming education [5, 9]. We denote a generated fixed program as $\mathcal{P}_\text{f}$, a generated explanation as $\mathcal{X}$, and generated feedback as a tuple $(\mathcal{P}_\text{f}, \mathcal{X})$.

Feedback quality. We assess the quality of generated feedback $(\mathcal{P}_\text{f}, \mathcal{X})$ w.r.t. $\mathcal{P}_\text{b}$ along the following binary attributes: (i) $\mathcal{P}_\text{f}$ is syntactically correct and is obtained by making a small number of edits to fix $\mathcal{P}_\text{b}$; (ii) $\mathcal{X}$ is complete, i.e., contains information about all errors and required fixes; (iii) $\mathcal{X}$ is correct, i.e., the provided information correctly explains errors and required fixes; (iv) $\mathcal{X}$ is comprehensible, i.e., easy to understand, presented in a readable format, and doesn't contain redundant information. These attributes are inspired by evaluation rubrics used in literature [6, 25–27]. In our evaluation, feedback quality is evaluated via ratings by experts along these four attributes. We measure feedback quality as binary by assigning the value of 1 (good quality) if it satisfies *all* the four quality attributes and otherwise 0 (bad quality).[2]

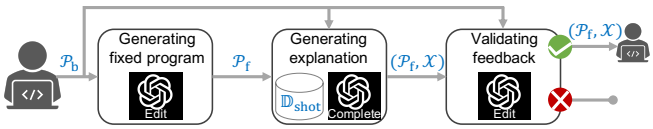


Figure 4: Illustration of three different compoments/stages in PyFiXV's feedback generation process; see Section 3.

2.2 Performance Metrics and Objective

Performance metrics. Next, we describe the overall performance metrics used to evaluate a feedback generation technique. For a buggy program $\mathcal{P}_\text{b}$ as input, we seek to design techniques that generate feedback $(\mathcal{P}_\text{f}, \mathcal{X})$ and also decide whether the generated feedback is suitable for sharing with the student. We measure the performance of a technique using two metrics: (i) *Coverage* measuring the percentage number of times the feedback is *generated and provided to the student*; (ii) *Precision* measuring the percentage number of times the *provided feedback is of good quality* w.r.t. the binary feedback quality criterion introduced above. In our experiments, we will compute these metrics on a dataset $\mathbb{D}_\text{test} = \{\mathcal{P}_\text{b}\}$ comprising a set of students' buggy programs.[3]

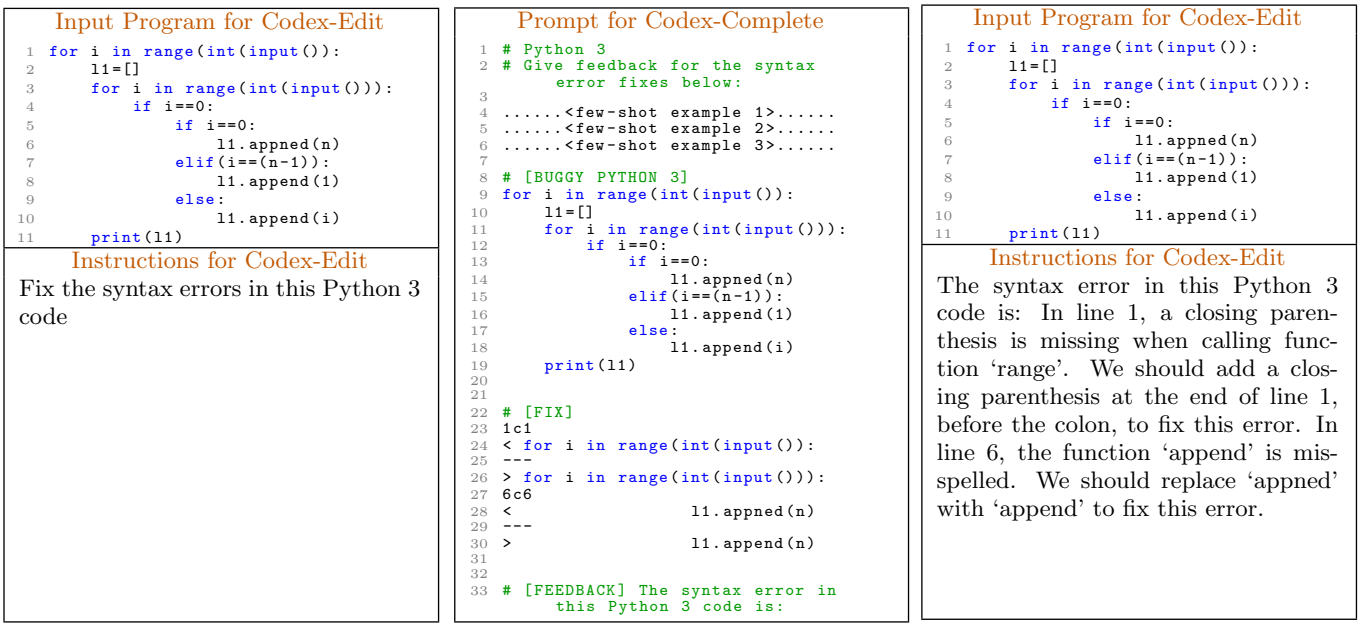
Objective. Our goal is to design feedback generation techniques with high precision, which is imperative before deploying such techniques in classrooms. In particular, we want to develop techniques with a tuneable precision parameter that could provide a knob to educators to control the precision and coverage trade-off.

3. OUR TECHNIQUE PyFiXV

In this section, we present PyFiXV, our technique to generate high-precision feedback using LLMs. PyFiXV uses OpenAPI's Codex as LLMs [1] – Codex has shown competitive performance on a variety of programming benchmarks [1, 3, 17, 18], and is particularly suitable for PyFiXV as we seek to generate both fixed programs and natural language explanations. More specifically, PyFiXV uses two access points of Codex provided by OpenAI through public APIs: Codex-Edit [28] and Codex-Complete [29]. As illustrated in Figure 4, PyFiXV has the following three components/stages: (1) generating a fixed program $\mathcal{P}_\text{f}$ by editing $\mathcal{P}_\text{b}$ using Codex-Edit; (2) generating natural language explanation $\mathcal{X}$ using Codex-Complete; (3) validating feedback $(\mathcal{P}_\text{f}, \mathcal{X})$ using Codex-Edit to decide whether the generated feedback is suitable for sharing. The overall pipeline of PyFiXV is modular and we will evaluate the utility of different components in Section 4. Next, we provide details for each of these stages.

[2]We note that the four attributes are independent. In particular, the attribute "complete" captures whether the explanation contains information about all errors/fixes (even though the information could be wrong), and the attribute "correct" captures the correctness of the provided information.

[3]When a technique cannot generate feedback for an input program $\mathcal{P}_\text{b}$ (e.g., the technique is unable to find a fixed program), then we use a natural convention that no feedback is provided to the student—this convention lowers the coverage metric but doesn't directly affect the precision metric.



(a) Stage-1 prompt for generating $\mathcal{P}_f$ (b) Stage-2 prompt for generating $\mathcal{X}$ (c) Stage-3 prompt for validating $(\mathcal{P}_f, \mathcal{X})$

Figure 5: Illustration of prompts used by different stages of PyFiXV for buggy Python 3 program in Figure 2. In particular, the "Instructions for Codex-Edit" in **(c)** is obtained by concatenating line33 of **(b)** and the generated $\mathcal{X}$ shown in Figure 2c.

3.1 Stage-1: Generating Fixed Program

Given a student's buggy program $\mathcal{P}_b$ as input, PyFiXV's Stage-1 generates a fixed program $\mathcal{P}_f$. We use Codex-Edit for fixing/repairing the buggy program in this stage since it has shown to be competitive in program repair benchmarks in recent works [30]. Figure 5a shows a sample prompt used by PyFiXV to query Codex-Edit for the buggy Python 3 program in Figure 2a. The process of generating $\mathcal{P}_f$ is determined by two hyperparameters: (i) $t_1 \in [0.0, 1.0]$ is the temperature value specified when querying Codex-Edit and controls stochasticity/diversity in generated programs; (ii) n_1 controls the number of queries made to Codex-Edit.

More concretely, PyFiXV begins by making n_1 queries to Codex-Edit with temperature t_1. Then, out of n_1 generated programs, PyFiXV selects $\mathcal{P}_f$ as the program that is syntactically correct and has the smallest *edit-distance* to $\mathcal{P}_b$. Here, edit-distance between two programs is measured by first tokenizing programs using Pygments library [31] and then computing Levenshtein edit-distance over token strings.[4] If Stage-1 is unable to generate a fixed program, the process stops without generating any feedback; see Footnote 3. In our experiments, we set $(t_1 = 0.5, n_1 = 10)$ and obtained a high success rate of generating a fixed program $\mathcal{P}_f$ with a small number of edits w.r.t. $\mathcal{P}_b$.

3.2 Stage-2: Generating Explanation

Given $\mathcal{P}_b$ and $\mathcal{P}_f$ as inputs, PyFiXV's Stage-2 generates a natural language explanation $\mathcal{X}$ describing errors/fixes. We use Codex-Complete in this stage as it is naturally suited to generate text by completing a prompt [1, 5, 6]. A crucial ingredient of Stage-2 is the annotated dataset $\mathbb{D}_{\text{shot}}$ used to select few-shot examples when querying Codex-Complete (see Figure 4). Figure 5b shows a sample prompt used by PyFiXV to query Codex-Complete for the scenario in Figure 2. In Figure 5b, line4–line6 indicate three few-shot examples (not shown for conciseness), line9–line19 provides $\mathcal{P}_b$, line23–line30 provides $\mathcal{P}_f$ in the form of line-diff w.r.t. $\mathcal{P}_b$, and line33 is the instruction to be completed by Codex-Complete. Given a prompt, the process of generating $\mathcal{X}$ is determined by two hyperparameters: (i) a temperature value t_2 $(= 0)$ and (ii) the number of queries n_2 $(= 1)$. Next, we discuss the role of $\mathbb{D}_{\text{shot}}$ in selecting few-shots examples.

When querying Codex-Complete, we use three few-shot examples selected from $\mathbb{D}_{\text{shot}}$, an annotated dataset of examples comprising buggy programs and desired feedback obtained by expert annotations (see Section 4.2). These annotated examples essentially provide a context to LLMs and have shown to play an important role in optimizing the generated output (e.g., see [1, 2, 17, 18, 32]). In our case, $\mathbb{D}_{\text{shot}}$ provides contextualized training data, capturing the format of how experts/tutors give explanations. Given $\mathcal{P}_b$ and $\mathcal{P}_f$, we use two main criteria to select few-shot examples. The primary criterion is to pick examples where the error type of buggy program in the example is same as that of $\mathcal{P}_b$—the underlying parser/compiler provides error types (e.g., 'InvalidSyntax', 'UnexpectedIndent'). The secondary criterion (used to break ties in the selection process) is based on the edit-distance *between* the *diff* of buggy/fixed program in the example and *diff* of $\mathcal{P}_b/\mathcal{P}_f$. In Section 4, we conduct ablations to showcase the importance of selecting few-shots.

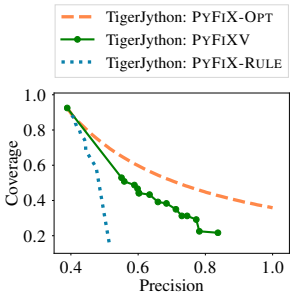
3.3 Stage-3: Validating Feedback

Given $\mathcal{P}_b$ and $(\mathcal{P}_f, \mathcal{X})$ as inputs, PyFiXV's Stage-3 validates the feedback quality and makes a binary decision of

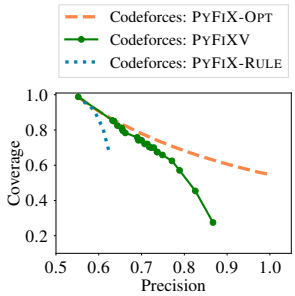
[4]Note that buggy programs are not parseable to *Abstract Syntax Tree* (AST) representations and string-based distance is commonly used in such settings (e.g., see [17]).

Technique	TigerJython		Codeforces	
	Precision	Coverage	Precision	Coverage
PyFi-PEM	05.0 (1.0)	92.5 (1.6)	35.0 (2.4)	98.8 (0.8)
PyFiX$_{\text{shot:None}}$	00.9 (0.5)	92.5 (1.6)	03.0 (0.4)	98.8 (0.8)
PyFiX$_{\text{shot:Rand}}$	21.6 (1.7)	92.5 (1.6)	48.5 (2.6)	98.8 (0.8)
PyFiX$_{\text{shot:Sel}}$	38.9 (3.5)	92.5 (1.6)	55.2 (3.9)	98.8 (0.8)
PyFi||X$_{\text{shot:Sel}}$	15.8 (1.8)	92.5 (1.6)	15.6 (2.8)	98.8 (0.8)
PyFiX-Rule$_{\text{P}\geq 70}$	48.6 (4.4)	30.8 (12.5)	61.6 (9.0)	38.3 (10.5)
PyFiXV$_{\text{P}\geq 70}$	76.0 (4.0)	31.2 (4.0)	72.4 (6.2)	64.2 (6.3)
PyFiX-Opt$_{\text{P}\approx V_{\text{P}\geq 70}}$	76.1 (0.4)	47.1 (3.4)	72.8 (0.1)	75.0 (5.7)

(a) Results for different techniques, reported as mean (stderr)



(b) TigerJython trade-off curve



(c) Codeforces trade-off curve

Figure 6: Experimental results on two real-world datasets of Python programs, namely TigerJython [9] and Codeforces [13].

"accept" (feedback is suitable for sharing) or "reject" (feedback is discarded). PyFiXV uses a novel run-time feedback validation mechanism using Codex-Edit to decide whether the feedback $(\mathcal{P}_{\text{f}}, \mathcal{X})$ w.r.t. $\mathcal{P}_{\text{b}}$ is of good quality. Here, Codex-Edit is used in the flipped role of a *simulated student model* – the intuition is that a good quality explanation $\mathcal{X}$, when provided in Codex-Edit's prompt instruction, should increase Codex-Edit's success in converting $\mathcal{P}_{\text{b}}$ to $\mathcal{P}_{\text{f}}$. Figure 5c shows a sample prompt used by PyFiXV to query Codex-Edit for the scenario in Figure 2—see the caption on how "Instructions for Codex-Edit" in Figure 5c is obtained.[5]

The validation mechanism has three hyperparameters: (i) $t_3 \in [0.0, 1.0]$ is the temperature value specified when querying Codex-Edit; (ii) n_3 controls the number of queries made to Codex-Edit; (iii) $h_3 \in [1, n_3]$ is the threshold used for acceptance decision. More concretely, PyFiXV begins by making n_3 queries to Codex-Edit with temperature t_3. Then, out of n_3 generated programs, PyFiXV counts the number of programs that don't have syntax errors and have an *exact-match* with $\mathcal{P}_{\text{f}}$. Here, exact-match is checked by converting programs to their *Abstract Syntax Tree* (AST)-based normalized representations.[6] Finally, the validation mechanism accepts the feedback if the number of exact matches is at least h_3. These hyperparameters (t_3, n_3, h_3) also provide a precision knob and are selected to obtain the desired precision level, as discussed next.

3.4 Precision and Coverage Trade-Off

PyFiXV's validation mechanism provides a precision knob to control the precision and coverage trade-off (see performance metrics in Section 2.2). Let P be the desired precision level we want to achieve for PyFiXV. The idea is to choose Stage-3 hyperparameters (t_3, n_3, h_3) that achieve P precision level. For this purpose, we use a calibration dataset $\mathbb{D}_{\text{cal}}$ for picking the hyperparameters. More concretely, in our experiments, PyFiXV first computes performance metrics on $\mathbb{D}_{\text{cal}}$ for the following range of values: (i) $t_3 \in \{0.3, 0.5, 0.8\}$; (ii) $n_3 \in \{10\}$; (iii) $h_3 \in \{1, 2, \ldots, 10\}$. Then, it chooses (t_3, n_3, h_3) that has at least P precision level and maximizes coverage; when achieving the desired P is not possible, then the next lower possible precision is considered. The chosen values of hyperparameters are then used in PyFiXV's Stage-3 validation mechanism. We refer to PyFiXV$_{\text{P}\geq x}$ as the version of PyFiXV calibrated with P $\geq x$.

[5]In our initial experiments, we tried using alternative signals for validation, such as (a) Codex-Complete's probabilities associated with generated $\mathcal{X}$; (b) automatic scoring of $\mathcal{X}$ w.r.t. explanations in few-shots using BLEU score [33]; (c) filtering based on $\mathcal{X}$'s length. Section 4 reports results for (c) as it had the highest performance among these alternatives.

[6]We check for AST-based exact match instead of checking for Levenshtein edit-distance over token strings being 0 (see Section 3.1). AST-based exact match is more relaxed than edit-distance being 0 – AST-based representation ignores certain differences between codes, e.g., based on extra spaces and comments. We used the AST-based exact match in the validation mechanism as it is more robust to such differences.

4. EXPERIMENTAL EVALUATION

We perform evaluations using two real-world Python programming datasets, namely TigerJython [9] and Codeforces [13]. We picked Python because of its growing popularity as an introductory programming language; notably, PyFiXV can be used with other languages by appropriately changing the prompts and tokenizers used. We use OpenAI's public APIs for Codex-Edit [28] (*model=code-davinci-edit-001*) and Codex-Complete [29] (*model=code-davinci-002*). We begin by describing different techniques used in the evaluation.

4.1 Baselines and Variants of PyFiXV

Default programming-error-messages without validation. As our first baseline, PyFi-PEM uses PyFiXV's Stage-1 to generate $\mathcal{P}_{\text{f}}$ and uses programming-error-messages provided by the programming environment as $\mathcal{X}$. PyFi-PEM uses error messages provided by Python 2.7 environment for TigerJython and Python 3.12 environment for Codeforces. This baseline is without validation (i.e., the generated feedback is always accepted).

Variants of PyFiXV without validation. PyFiX$_{\text{shot:Sel}}$ is a variant of PyFiXV without the validation mechanism (i.e., only uses Stage-1 and Stage-2). PyFiX$_{\text{shot:Rand}}$ is a variant of PyFiX$_{\text{shot:Sel}}$ where few-shot examples in Stage-2 are picked randomly from $\mathbb{D}_{\text{shot}}$. PyFiX$_{\text{shot:None}}$ is a variant of PyFiX$_{\text{shot:Sel}}$ that doesn't use few-shot examples in Stage-2. PyFi||X$_{\text{shot:Sel}}$ is a variant of PyFiX$_{\text{shot:Sel}}$ that runs Stage-1 and Stage-2 in parallel; hence, Stage-2's prompt doesn't make use of $\mathcal{P}_{\text{f}}$. All these variants are without validation (i.e., the generated feedback is always accepted).

Techniques with alternative validation mechanisms. We consider two variants of PyFiXV, namely PyFiX-Rule and PyFiX-Opt, that use different validation mechanisms (i.e., replace PyFiXV's Stage-3 with an alternative validation).

```python
import time

def cd(seconds):
    while seconds > 0:
        rem_m = seconds // 60
        rem_s = seconds % 60
        remaining_time = str(rem_m) + "
     minutes " + str(rem_s) + " seconds"
        print(remaining_time, end='\r')
        seconds = seconds - 1
        time.sleep(1)

    print("Xxxxx xxxxX")

cd(60)
```

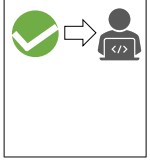
(a) Student's buggy program $\mathcal{P}_b$

```python
from __future__ import print_function
import time

def cd(seconds):
    while seconds > 0:
        rem_m = seconds // 60
        rem_s = seconds % 60
        remaining_time = str(rem_m) + "
     minutes " + str(rem_s) + " seconds"
        print(remaining_time, end='\r')
        seconds = seconds - 1
        time.sleep(1)

    print("Xxxxx xxxxX")

cd(60)
```

(b) Generated $\mathcal{P}_f$

The student wants to use the print function from Python 3. To do so, the correct way is to add the line 'from __future__ import print_function' at the beginning of the code.

(c) Generated $\mathcal{X}$



(d) Validation

Figure 7: Similar to Figure 1, this illustrative example showcases PyFiXV on a buggy Python 2 program from TigerJython [9].

```python
name = input("Xxx xx?")
num = input("Xxx xx xxxxxxx?")
print ("Xxxx " + name + "Xxx ")*,num
```

(a) Student's buggy program $\mathcal{P}_b$

```python
name = input("Xxx xx?")
num = input("Xxx xx xxxxxxx?")
print ("Xxxx " + name + "Xxx ")*num
```

(b) Generated $\mathcal{P}_f$

The student forgets to enclose a string literal with quotes. We can fix the error by enclosing the string literal in line 3 with a pair of double quotes.

(c) Generated $\mathcal{X}$

(d) Validation

Figure 8: Similar to Figure 3, this example showcases PyFiXV on a buggy Python 2 program from TigerJython [9]. PyFiXV's validation mechanism *successfully rejected* the generated feedback (we marked text in **(c)** to highlight issues with explanation).

PyFiX-Rule validates $(\mathcal{P}_f, \mathcal{X})$ based on $\mathcal{X}$'s length, as noted in Footnote 5. Given a hyperparameter h_r, $(\mathcal{P}_f, \mathcal{X})$ is accepted if the number of tokens in $\mathcal{X}$ is at most h_r, where tokenization is done by splitting on whitespaces/punctuations. PyFiX-Rule's h_r is picked from the set $\{30, 40, 50, \ldots, 200\}$ based on the desired precision level P, by following the calibration process in Section 3.4. PyFiX-Opt uses an oracle validation that has access to expert's ratings for the generated feedback $(\mathcal{P}_f, \mathcal{X})$. Then, for a desired P, PyFiX-Opt performs optimal validation and highlights the maximum coverage achievable on $\mathbb{D}_{\text{test}}$ for the generated feedback.

4.2 Datasets and Evaluation Procedure

Datasets and annotations for few-shot examples. As our first dataset, namely TigerJython, we have 240 distinct Python 2 programs written by students in TigerJython's educational programming environment [9]. We obtained a private and anonymized version of the dataset used in [34], with string literals in programs replaced with sequences of 'x' (e.g., see Figure 1). As our second dataset, namely Codeforces, we curated 240 distinct Python 3 programs from the Codeforces website using their public APIs [13], inspired by similar works that curate Codeforces dataset [35, 36]. Programs in both datasets have syntax errors and have token length at most 500 (see Section 3.1 about program tokenization). For the Codeforces dataset, we only include programs submitted to contests held from July 2021 onwards (after the cut-off date for Codex's training data [1]). Since a part of these datasets will be used for few-shot examples (as $\mathbb{D}_{\text{shot}}$ in PyFiXV's Stage-2), we asked experts to annotate these 480 programs with feedback (i.e., a fixed program along with an explanation). Three experts, with extensive experience in Python programming and tutoring, provided annotations.

Evaluation procedure and feedback ratings. Given a dataset $\mathbb{D}$ with 240 buggy programs, we can evaluate a technique by splitting $\mathbb{D}$ as follows: (a) $\mathbb{D}_{\text{test}}$ (25%) for reporting precision and coverage performance metrics; (b) $\mathbb{D}_{\text{shot}}$ (50%) for few-shot examples; (c) $\mathbb{D}_{\text{cal}}$ (25%) for calibrating validation mechanism. To report overall performance for techniques, we perform a cross-validation procedure with four evaluation rounds while ensuring that $\mathbb{D}_{\text{test}}$ across four rounds are non-overlapping. We then report aggregated results across these rounds as average mean (stderr). As discussed in Sections 2.1 and 2.2, evaluating these performance metrics requires feedback ratings by experts to assess the quality of the feedback generated by each technique.[7] For example, evaluating metrics on TigerJython dataset for PyFiXV requires 480 feedback ratings (4×60 for $\mathbb{D}_{\text{test}}$ and 4×60 for $\mathbb{D}_{\text{cal}}$). To begin, we did a smaller scale investigation to establish the rating criteria, where two experts rated 100 generated feedback instances; we obtained Cohen's kappa reliability value 0.72 indicating *substantial agreement* between experts [37]. Afterward, one expert (with experience in tutoring Python programming classes) did these feedback ratings for the evaluation results.[8]

4.3 Results

Comparison of different techniques. Figure 6a provides a comparison of different techniques on two datasets. All techniques here use PyFiXV's Stage-1 to obtain $\mathcal{P}_f$. The coverage numbers of 92.5 and 98.8 reported in Figure 6a correspond to the success rate of obtaining $\mathcal{P}_f$ on these datasets (the average edit-distance between $\mathcal{P}_b$ and $\mathcal{P}_f$ is about 10.4 and 7.5 tokens on these datasets, respectively). For our baseline PyFi-PEM, we see a big jump in precision from 5.0 for TigerJython (Python 2) to 35.0 for Codeforces (Python

[7] We note that precision and coverage performance metrics for different techniques are reported for the end-to-end process associated with each technique, and not just for the validation mechanism. Also, even if a technique doesn't use any validation mechanism, the coverage could be less than 100.0 as discussed in Footnote 3.

[8] We note that the experts were blinded to the condition (technique) associated with each feedback instance when providing ratings. Moreover, these generated feedback instances were given to experts in randomized order across conditions instead of grouping them per condition.

3), owing to enhanced error messages in recent Python versions [38–40]. Results for PyFiXV$_{P\geq 70}$ in comparison with results for PyFiX$_{\text{shot:Sel}}$, PyFiX$_{\text{shot:Rand}}$, PyFiX$_{\text{shot:None}}$, and PyFi||X$_{\text{shot:Sel}}$ showcase the utility of different components used in PyFiXV's pipeline. Comparing PyFiXV$_{P\geq 70}$ with PyFiX-Rule$_{P\geq 70}$ shows that PyFiXV's validation substantially outperforms PyFiX-Rule's validation.[9] Lastly, results for PyFiX-Opt$_{P\approx V_{P\geq 70}}$ are obtained by setting the desired precision level for PyFiX-Opt to match that of PyFiXV$_{P\geq 70}$ on $\mathbb{D}_{\text{test}}$ – the coverage numbers (47.1 for TigerJython and 75.0 for Codeforces) indicate the maximum possible achievable coverage. Notably, PyFiXV$_{P\geq 70}$ achieves a competitive coverage of 64.2 on Codeforces.[10]

Precision and coverage trade-off curves. The curves in Figures 6b and 6c are obtained by picking different desired precision levels P and then computing precision/coverage values on $\mathbb{D}_{\text{test}}$ w.r.t. P. The curves for PyFiX-Opt show the maximum possible coverage achievable on $\mathbb{D}_{\text{test}}$ for different precision levels P using our generated feedback. To obtain these curves for PyFiXV and PyFiX-Rule, we did calibration directly on $\mathbb{D}_{\text{test}}$ instead of $\mathbb{D}_{\text{cal}}$ (i.e., doing ideal calibration for their validation mechanisms when comparing with PyFiX-Opt's curves). These curves highlight the precision and coverage trade-off offered by PyFiXV in comparison to a simple rule-based validation and the oracle validation.

Qualitative analysis. We have provided several illustrative examples to demonstrate our technique PyFiXV. Figures 1, 2, and 7 show examples where PyFiXV's Stage-1 and Stage-2 generate good quality feedback and Stage-3 successfully accepts the feedback. Figures 3 and 8 show examples where PyFiXV's Stage-1 and Stage-2 generate bad quality feedback and Stage-3 successfully rejects the feedback. Figure 7 highlights that PyFiXV can make non-trivial fixes in the buggy program and correctly explain them in a comprehensible way. Figure 3 shows an example where the overall feedback is bad quality and successfully rejected, though parts of the generated explanation are correct; this could potentially be useful for tutors in a human-in-the-loop approach.

5. CONCLUDING DISCUSSIONS

We investigated using LLMs to generate feedback for fixing programming syntax errors. In particular, we considered feedback in the form of a fixed program along with a natural language explanation. We focussed on the challenge of generating high-precision feedback, which is crucial before deploying such technology in classrooms. Our proposed technique, PyFiXV, ensures high precision through a novel run-time validation mechanism and also provides a precision knob to educators. We performed an extensive evaluation to showcase the efficacy of PyFiXV on two real-world Python programming datasets. There are several interesting directions for future work, including (a) improving PyFiXV's components to obtain better precision/coverage trade-off, e.g., by adapting our technique to use recent LLMs such as ChatGPT [42] and GPT-4 [43] instead of Codex; (b) extending PyFiXV beyond syntax errors to provide feedback for programs with semantic errors or partial programs; (c) incorporating additional signals in PyFiXV's validation mechanism; (d) conducting real-world studies in classrooms.

[9]When comparing PyFiXV$_{P\geq 70}$ with these techniques in Figure 6a, the results are significantly different w.r.t. χ^2 tests [41] ($p \leq 0.0001$); here, we use contingency tables with two rows (techniques) and four columns (240 data points mapped to four possible precision/coverage outcomes).

[10]Techniques PyFiX$_{\text{shot:Sel}}$, PyFiX-Rule, PyFiXV$_{P\geq 70}$, and PyFiX-Opt$_{P\approx V_{P\geq 70}}$ differ only in terms of validation mechanisms. We can compare the validation mechanisms used in these techniques based on F1-score. The F1-scores of these four techniques are as follows: 0.56, 0.39, 0.70, and 0.86 for TigerJython, respectively; 0.71, 0.47, 0.77, and 0.84 for Codeforces, respectively.

6. ACKNOWLEDGMENTS

Funded/Co-funded by the European Union (ERC, TOPS, 101039090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Mark Chen and et al. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374, 2021.

[2] Tom B. Brown and et al. Language Models are Few-Shot Learners. In *NeurIPS*, 2020.

[3] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *ACE*, 2022.

[4] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *ICER*, 2022.

[5] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *SIGCSE*, 2023.

[6] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent N. Reeves, Paul Denny, James Prather, and Brett A. Becker. Using Large Language Models to Enhance Programming Error Messages. In *SIGCSE*, 2023.

[7] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani L. Peters, John Homer, Nevan Simone, and Maxine S. Cohen. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *ICER*, 2017.

[8] Brett A. Becker. An Effective Approach to Enhancing Compiler Error Messages. In *SIGCSE*, 2016.

[9] Tobias Kohn and Bill Z. Manaris. Tell Me What's Wrong: A Python IDE with Error Messages. In *SIGCSE*, 2020.

[10] Brett A. Becker. What Does Saying That 'Programming is Hard' Really Say, and About Whom? *Communications of ACM*, 64(8):27–29, 2021.

[11] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*, 2013.

[12] Samim Mirhosseini, Austin Z. Henley, and Chris Parnin. What is Your Biggest Pain Point? An Investigation of CS Instructor Obstacles, Workarounds, and Desires. In *SIGCSE*, 2023.

[13] Mikhail Mirzayanov. Codeforces. `https://codeforces.com/`.

[14] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated Clustering and Program Repair for Introductory Programming Assignments. In *PLDI*, 2018.

[15] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *ICSE*, 2018.

[16] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. In *AAAI*, 2019.

[17] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing Bugs in Python Assignments Using Large Language Models. *CoRR*, abs/2209.14876, 2022.

[18] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. Repair is Nearly Generation: Multilingual Program Repair with LLMs. In *AAAI*, 2023.

[19] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *CHI*, 2010.

[20] Andrew Head, Elena L. 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 *Learning @ Scale*, 2017.

[21] Darren Key, Wen-Ding Li, and Kevin Ellis. I Speak, You Verify: Toward Trustworthy Neural Program Synthesis. *CoRR*, abs/2210.00848, 2022.

[22] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding Back-Translation at Scale. In *EMNLP*, 2018.

[23] Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. Program Synthesis with Pragmatic Communication. In *NeurIPS*, 2020.

[24] Hiroaki Funayama, Tasuku Sato, Yuichiroh Matsubayashi, Tomoya Mizumoto, Jun Suzuki, and Kentaro Inui. Balancing Cost and Quality: An Exploration of Human-in-the-Loop Frameworks for Automated Short Answer Scoring. In *AIED*, 2022.

[25] Rui Zhi, Samiha Marwan, Yihuan Dong, Nicholas Lytle, Thomas W. Price, and Tiffany Barnes. Toward Data-Driven Example Feedback for Novice Programming. In *EDM*, 2019.

[26] Ahana Ghosh, Sebastian Tschiatschek, Sam Devlin, and Adish Singla. Adaptive Scaffolding in Block-Based Programming via Synthesizing New Tasks as Pop Quizzes. In *AIED*, 2022.

[27] Anaïs Tack and Chris Piech. The AI Teacher Test: Measuring the Pedagogical Ability of Blender and GPT-3 in Educational Dialogues. In *EDM*, 2023.

[28] OpenAI. Codex-Edit. `https://beta.openai.com/playground?mode=edit&model=code-davinci-edit-001`, .

[29] OpenAI. Codex-Ccomplete. `https://beta.openai.com/playground?mode=complete&model=code-davinci-002`, .

[30] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. Automated Repair of Programs from Large Language Models. In *ICSE*, 2022.

[31] Georg Brandl, Matthäus Chajdas, and Jean Abou-Samra. Pygments. `https://pygments.org/`.

[32] Rohan Bavishi, Harshit Joshi, José Cambronero, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radicek, and Ashish Tiwari. Neurosymbolic Repair for Low-Code Formula Languages. *Proceedings ACM Programming Languages*, 6(OOPSLA2), 2022.

[33] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *ACL*, 2002.

[34] Tobias Kohn. The Error Behind The Message: Finding the Cause of Error Messages in Python. In *SIGCSE*, 2019.

[35] Ethan Caballero and Ilya Sutskever. Description2Code Dataset. `https://github.com/ethancaballero/description2code`, 2016.

[36] Yujia Li and et al. Competition-Level Code Generation with AlphaCode. 2022.

[37] Matthijs J Warrens. Five Ways to Look at Cohen's Kappa. *Journal of Psychology & Psychotherapy*, 5(4): 1, 2015.

[38] The Python Software Foundation. What's New In Python 3.10. `https://docs.python.org/3/whatsnew/3.10.html`, .

[39] The Python Software Foundation. What's New In Python 3.11. `https://docs.python.org/3/whatsnew/3.11.html`, .

[40] The Python Software Foundation. What's New In Python 3.12. `https://docs.python.org/3.12/whatsnew/3.12.html`, .

[41] William G Cochran. The $\chi 2$ Test of Goodness of Fit. *The Annals of Mathematical Statistics*, 1952.

[42] OpenAI. ChatGPT. `https://openai.com/blog/chatgpt`, 2023.

[43] OpenAI. GPT-4 Technical Report. *CoRR*, abs/2303.08774, 2023.