PySTAAR: An End-to-End, Extensible Framework for Automated Python Type Error Repair

Wonseok Oh
Korea University
Seoul, Republic of Korea
wonseok_oh@korea.ac.kr

Hyobin Park
Kyungpook National University
Daegu, Republic of Korea
guardian312@knu.ac.kr

Miryeong Kang
Korea University
Seoul, Republic of Korea
miryeong59@korea.ac.kr

Seungbin Choi
Kyungpook National University
Daegu, Republic of Korea
csb8226@naver.com

Yunja Choi
Kyungpook National University
Daegu, Republic of Korea
kaisertoto@gmail.com

Hakjoo Oh
Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

Abstract

We present PySTAAR, an end-to-end, extensible framework for automatically detecting and repairing type errors in Python programs. PySTAAR integrates test generation, fault localization, patch synthesis, and patch validation into a fully automated pipeline, leveraging large language models and state-of-the-art repair techniques. The framework features a modular, extensible architecture and a user-friendly web interface that visualizes the workflow, making it accessible to developers of all experience levels. We demonstrate that PySTAAR can automatically detect and repair type errors in real-world Python applications, achieving high repair rate without manual intervention. Our tool demonstration is available at https://youtu.be/VizRQsrtsdk.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Python, Type Error, Automated Program Repair

ACM Reference Format:
Wonseok Oh, Hyobin Park, Miryeong Kang, Seungbin Choi, Yunja Choi, and Hakjoo Oh. 2026. PySTAAR: An End-to-End, Extensible Framework for Automated Python Type Error Repair. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3774748.3787613

1 Introduction

Python is currently the most popular programming language in the world. According to the 2024 GitHub Octoverse report [4], Python has ranked as the most widely used and fastest-growing programming language. The popularity is largely due to its versatility and flexibility, making it a language of choice for a wide range of applications, from web development to machine learning.

Python's flexibility stems from its dynamic type system, where variables can hold values of any type without requiring explicit type declarations. This not only simplifies the language syntax but also enables powerful features such as duck typing, dynamic dispatch, metaprogramming, and flexible higher-order functions, all of which facilitate developer productivity and rapid prototyping.

However, this flexibility comes at a cost: type errors can occur at runtime when variables are used in ways incompatible with their actual types. These errors are among the most frustrating and time-consuming bugs for Python developers. A recent study [12] found that they account for nearly 30% of all built-in exceptions in real-world Python projects and are often difficult to diagnose and fix, sometimes remaining unresolved for weeks or even months.

To address these challenges, we present PySTAAR[1], a framework for automatically detecting and repairing type errors in Python programs. PySTAAR offers three key features:

- **Fully automated**: PySTAAR is an end-to-end, fully automated tool that unifies test generation, fault localization, patch synthesis, and patch validation into a single pipeline, enabling developers to identify and fix type errors without manual intervention. To this end, it leverages large language models (LLMs) for targeted test generation and state-of-the-art techniques [12] to produce high-quality patches.
- **Extensible**: The framework features a modular, extensible architecture that allows users to easily swap out components for different error detection and repair techniques.
- **User-friendly**: PySTAAR includes an easy-to-use web interface, allowing users to upload Python programs and receive patches for detected type errors, making the tool accessible to developers of all experience levels.

Intended Users. PySTAAR is designed for both practitioners and researchers. First, it serves as a practical end-to-end tool for beginner- to intermediate-level developers, especially in industry settings lacking high-quality test cases or expertise in diagnosing and fixing type errors. In such cases, PySTAAR can significantly reduce the manual effort required to address those errors.

Second, PySTAAR supports researchers working on Python error detection and repair. The framework is explicitly designed for extensibility, enabling researchers to focus on developing individual components such as test generation, fault localization, or support

[1] **Py**thon **S**pecialized **T**ype **A**nalysis & **A**utomatic **R**epair.

ICSE-Companion '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3787613

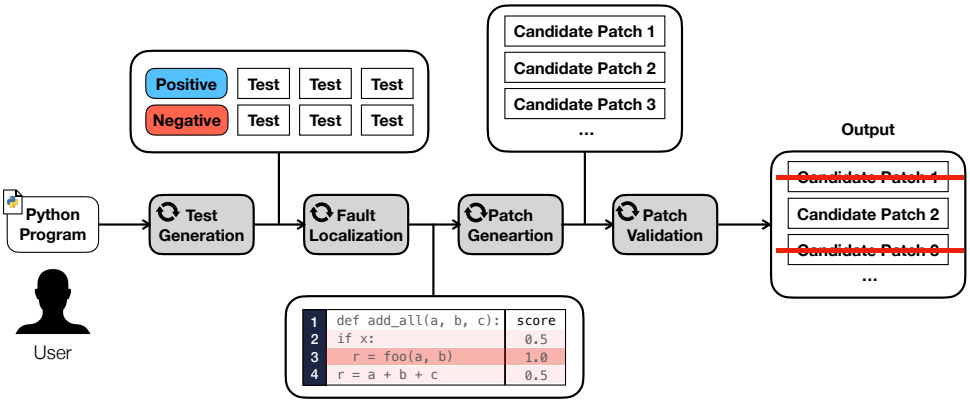

Figure 1: Architecture of PySTAAR. Each step shaded in gray is designed to be extensible.

for additional error types. In this way, PySTAAR provides a reusable infrastructure that facilitates program repair research for Python.

Contributions and Positioning. Most core techniques used by PySTAAR—including fault localization, patch generation, and patch validation—are based on our prior work, PyTER [12]. The contribution of this paper is to extend PyTER into a fully automated, extensible, and user-friendly tool. To this end, we add automatic test generation powered by LLMs, refactor each component of PyTER into modular units, and develop a web interface.

No existing tool offers the integrated infrastructure that PySTAAR provides. While many approaches address detecting or repairing Python type errors [1, 3, 5, 6, 9–13, 15, 16, 19], they are typically limited to specific stages of the process (e.g., fault localization and patch generation), requiring developers to rely on multiple tools. In contrast, PySTAAR delivers a practical, accessible solution even for developers with limited testing or debugging experience.

Moreover, most existing tools are not designed for extensibility, making it difficult to customize or extend them with new techniques. FixKit [18], a Python library for automatic program repair, is an exception, as it supports the integration and replacement of different repair techniques. However, FixKit relies on user-provided test cases to detect errors, whereas PySTAAR automates the entire process, including test case generation for identifying type errors.

Tool Availability. PySTAAR is publicly available:

- Source Code: https://github.com/kupl/Pystaar
- Web interface: https://pystaar.org
- Demonstration video: https://youtu.be/VizRQsrtsdk

2 Design and Implementation

2.1 System Architecture

Figure 1 presents the overall architecture of PySTAAR. Once a user provides a Python program, PySTAAR executes the following pipeline in a fully automated manner: (1) **Test case generation** for both successful (positive) and type-error-triggering (negative) executions, (2) **Fault localization** to identify the target line for repair (3) **Patch generation** to synthesize candidate fixes, and (4) **Patch validation** to verify the correctness of generated patches.

We employ a modular design in which components exchange inputs and outputs via standardized formats (e.g., JSON or Python schema). This design enables researchers to easily substitute components with their own implementations by adhering to the standard interfaces, facilitating seamless integration of new techniques without modifying the rest of the framework. Each component also supports standalone execution, allowing for independent testing and use. By leveraging this extensibility, researchers can readily quantify the impact of their techniques within the full pipeline.

2.2 Core Components

PySTAAR consists of modular components that collectively form a practical solution for Python type errors. It leverages large language models (LLMs) for test case generation, and employs PyTER [12] to perform fault localization, patch generation, and patch validation.

Test Case Generation. Our framework begins with detecting type errors and generating test cases using LLMs. The test generation consists of two main steps: (1) identifying lines that are likely to trigger type errors, and (2) generating both positive and negative test cases to cover the identified lines.

First, PySTAAR takes a Python file (or function) as input and instructs an LLM (GPT-4o [14]) to identify suspicious lines that are likely to cause type errors. It then prompts the LLM to generate positive and negative test cases that cover the identified lines, with the number of covered lines configurable by the user. For each identified line, PySTAAR produces corresponding positive and negative tests for valuable coverage [2, 7, 17].

Fault Localization. With the generated test cases, PySTAAR executes them to collect execution traces and identify the suspicious locations that may require modification to repair type errors. PySTAAR employs a fault localization technique specialized for type errors [12]. This process first ranks the functions that are likely to contain type errors and then pinpoints the suspicious lines within those functions, ultimately identifying variables that require repair.

Patch Generation. Given the suspicious lines and variables, PySTAAR generates candidate patches for the detected type errors. We utilize type-specialized patch templates designed to address common type error patterns, such as changing a variable's type or

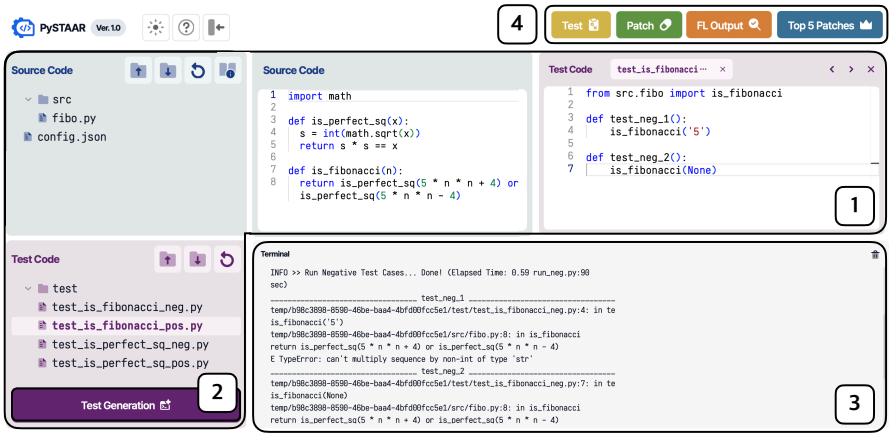

Figure 2: Web interface of PySTAAR. Initially, the 'Test' and 'Patch' buttons are deactivated. 'Test' is activated after tests are generated. 'Patch' is activated after pressing the Test button. 'FL Output' is activated after pressing 'Patch'.

inserting type-check statements. As a result, PySTAAR produces candidate patches that can repair the identified type errors.

Patch Validation. After generating candidate patches, PySTAAR validates their correctness by executing each patch against the test cases to determine whether the type errors are resolved. Patches that eliminate the errors are considered plausible; PySTAAR then verifies whether each plausible patch preserves the original function's return type. Only patches that satisfying both criteria are considered valid and returned as the final output.

2.3 Command-Line Interface

We provide a command-line interface (CLI) that allows users to run the framework locally or integrate it into their existing workflows. To support diverse environments, PySTAAR provides both Docker-based and local installation options. The entire framework can be executed with the following command:

```
python pystaar.py
    --source-file <src_file_path>
    --source-directory <src_root_directory>
    --project-directory <project_directory>
```

Here, `src_file_path` refers to the target Python file to be analyzed, `source_directory` the source code's root directory, and `project_directory` the top-level project path. In addition, we support commands and script files for running each component independently. For detailed setup and usage instructions, refer to the README at https://github.com/kupl/Pystaar.

3 Web Interface

To assist novice developers, PySTAAR also provides a web-based interface. Figure 2 presents its four main components: (1) a **Code Editor** for viewing and editing source and test files, supporting direct code uploads and automatic generation of `config.json` and `requirements.txt` files upon project upload; (2) a **Test Generation Panel** for generating test cases for selected code with configurable options; (3) a **Terminal Output** panel that displays real-time results of test execution and patch generation; and (4) a **Control Toolbar** for running tests, trigger automated patch generation, and inspecting fault localization output and top patch candidates.

Table 1: Evaluation on 7 samples from CrowdQuake [8]. #N and #P denote the numbers of negative and positive test cases generated for each type error, respectively. #Gen and #Val denote the numbers of generated and validated patches, respectively. 'Patched' indicates whether the type error was successfully repaired (✓) or not (✗).

#	Test Gen		Patch Information		
	#N	#P	#Gen	#Val	Patched
#1	3	7	112	48	✓
#2	3	0	42	0	✓
#3	3	5	124	2	✓
#4	3	5	258	26	✓
#5	3	5	424	20	✓
#6	3	5	72	0	✗
#7	3	5	267	24	✓

4 Evaluation

We evaluated the usefulness of PySTAAR in both industrial and open-source contexts. All experiments were conducted on a machine with an 8-core Apple M3 CPU with 16GB of RAM, and the scripts for reproduction are publicly available in our repository[2].

Industrial Application. We applied PySTAAR to CrowdQuake [8], a distributed seismic monitoring system developed by Kyungpook National University with support from Korea Meteorological Administration and SK Telecom, deployed in South Korea. Our evaluation targeted its data analysis component, consisting of 248 Python

[2]https://github.com/kupl/Pystaar

Table 2: Evaluation on open-source Python projects. #Gen and #Val denote the numbers of generated and validated patches, respectively. 'Patched' indicates whether the type error was successfully repaired (✓) or not (✗). The last four columns show the time taken for each phase of PySTAAR.

Projects		Patch Information			Time			
Name	kLoc	#Gen	#Val	Patched	Running Tests	Localization	Generation	Validation
requests	5.8	262	6	✓	7.49s	0.46s	29.28s	1m04.01s
pandas	300.9	2083	26	✓	9m01.99s	0.25s	22m31.33s	9m28.48s
salt	519.6	4660	6	✓	11.23s	0.23s	17m21.28s	9m25.98s

files and 16,316 lines of code. Although the system has been under development for six years, testing remains limited due to tight schedules, budget constraints, and a shortage of experienced programmers. Consequently, no predefined test suite exists, and errors are typically detected and fixed during operation. Running PySTAAR took 49.8s on average, automatically detecting 49 type errors and successfully repairing 46, resulting in a 93.9% fix rate.

While the full source code of CrowdQuake is publicly unavailable, we obtained permission from the developers to share a subset containing representative type errors. Table 1 presents the results of applying PySTAAR to these examples, demonstrating that PySTAAR can effectively handle industrial projects with limited testing infrastructure. These code examples are available in our repository.

Open-Source Applications. We also evaluated PySTAAR on known type errors from open-source Python projects using the TypeBugs benchmark [12]. We selected three projects of varying sizes (5.8–519.6kLoC)—`requests`[3], `pandas`[4], and `salt`[5]—and used the most recent type error from each as the evaluation target.

Table 2 summarizes the results of our evaluation. PySTAAR successfully repaired type errors in all three projects within a reasonable time. Notably, it produced a top-1 patch for the `pandas` and `salt` projects, and a top-5 patch for the `requests` project. The results demonstrate that PySTAAR can effectively handle real-world Python projects with varying sizes and complexities. Details of the bugs and generated patches are provided in our repository.

5 Conclusion

We presented PySTAAR, an end-to-end, extensible framework for automatically detecting and repairing type errors in Python. It integrates LLM-based test generation, fault localization, patch synthesis, and validation into a fully automated pipeline, featuring a modular architecture and a user-friendly web interface. To demonstrate its practical utility, we applied PySTAAR to three popular open-source Python projects and an industrial Python application. We hope PySTAAR serves as both a practical tool and an extensible platform for developers and researchers.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2021R1A5A1021944, 70%), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, 25%), and ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2025-RS-2020-II201819, 5%).

[3]https://github.com/psf/requests
[4]https://github.com/pandas-dev/pandas
[5]https://github.com/saltstack/salt

References

[1] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *IEEE/ACM International Conference on Software Engineering*.
[2] Stephen H Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests?. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 171–176.
[3] facebook. 2017. Pyre-check: Performant type-checking for python. https://github.com/facebook/pyre-check
[4] github. 2024. Octoverse: AI leads Python to top language as the number of global developers surges. https://github.blog/news-insights/octoverse/octoverse-2024/
[5] google. 2015. Pytype: A static type analyzer for Python code. https://github.com/google/pytype
[6] Sichong Hao, Xianjun Shi, and Hongwei Liu. 2024. RetypeR: Integrated Retrieval-based Automatic Program Repair for Python Type Errors. In *International Conference on Software Maintenance and Evolution*.
[7] Andre Hora. 2024. Test polarity: detecting positive and negative tests. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 537–541.
[8] Xin Huang, Jangsoo Lee, Young-Woo Kwon, and Chul-Ho Lee. 2020. CrowdQuake: A Networked System of Low-Cost Sensors for Earthquake Detection via Deep Learning. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.
[9] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for Python. In *ACM/IEEE International Conference on Software Engineering: Companion Proceedings*.
[10] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (2023), 36.
[11] microsoft. 2019. Pyright: Static Type Checker for Python. https://github.com/microsoft/pyright
[12] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[13] Wonseok Oh and Hakjoo Oh. 2024. Towards Effective Static Type-Error Detection for Python. In *IEEE/ACM International Conference on Automated Software Engineering*.
[14] OpenAI. 2025. GPT-4o. https://platform.openai.com/docs/models/gpt-4o.
[15] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors. In *IEEE/ACM International Conference on Software Engineering*.
[16] python. 2012. Mypy: Static Typing for Python. https://github.com/python/mypy
[17] Iflaah Salman, Burak Turhan, and Sira Vegas. 2019. A controlled experiment on time pressure and confirmation bias in functional software testing. *Empirical Software Engineering* 24, 4 (2019), 1727–1761.
[18] Marius Smytzek, Martin Eberlein, Kai Werk, Lars Grunske, and Andreas Zeller. 2024. FixKit: A Program Repair Collection for Python. In *IEEE/ACM International Conference on Automated Software Engineering*.
[19] Ruofan Yang, Xianghua Xu, and Ran Wang. 2025. LLM-enhanced evolutionary test generation for untyped languages. *Automated Software Engineering* 32, 1 (2025), 20.