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. 2018. PySTAAR: An End-to-End, Extensible Framework for Automated Python Type Error Repair. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Python is currently the most popular programming language in the world. According to the 2024 GitHub Octoverse report [3], Python has ranked as the most widely used and fastest-growing programming language, supported by a large and active community of developers. 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 largely 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 (e.g., by eliminating the need to annotate variable types) 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 [10] 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 integrates test generation, fault localization, patch synthesis, and patch validation into a unified pipeline, enabling developers to identify and fix type errors without manual intervention. To this end, it leverages large language models (LLMs) to generate targeted test cases and builds upon state-of-the-art techniques [10] to produce high-quality patches.
- **Extensible**: The framework is designed to be extensible, featuring a modular 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 can serve as a practical end-to-end tool for beginner- to intermediate-level developers, especially in industry settings where high-quality test cases may be unavailable and developers may be unfamiliar with type errors or how to fix them. In

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

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

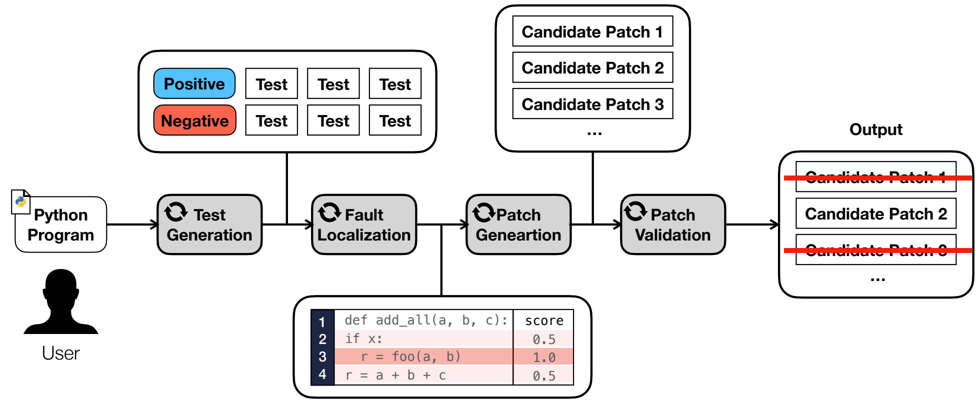

Figure 1: Architecture of PySTAAR. Each step shaded in gray is designed to be extensible.

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 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 [10]. 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 tools and techniques exist for detecting or repairing Python type errors [1, 2, 4, 5, 7–11, 13, 14, 16], they typically focus on a single aspect, forcing developers to combine multiple tools. This fragmented approach is cumbersome, error-prone, and inefficient. In contrast, PySTAAR delivers a practical, accessible solution even for developers with little testing or debugging experience.

Moreover, most existing tools are not designed with extensibility in mind, making it difficult for researchers or practitioners to customize or extend them with new techniques. An exception is FixKit [15], a Python library for automatic program repair that allows easy integration and replacement of different repair techniques. However, FixKit requires 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 determine a line to be modified to repair the type error, (3) **Patch generation** to synthesize candidate fixes, and (4) **Patch validation** to ensure correctness of the generated patches.

We employ a modular design in which components exchange inputs and outputs using standardized formats (e.g., JSON or Python schema). This architecture allows researchers to easily substitute components with their own implementations by simply adhering to the standard interfaces, enabling seamless integration of new techniques without changes to the rest of the framework. Additionally, each component is designed for 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, each responsible for a core functionality, collectively forming a practical solution for automated Python type error repair. We leverage large language models (LLMs) to generate test cases, and employ PyTER [10] to implement the fault localization, patch generation, and patch validation components.

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.

At first, PySTAAR takes a Python file (or function) as input and instructs an LLM (GPT-4o [12]) to identify suspicious lines that are likely to cause type errors. Next, PySTAAR prompts the LLM to generate test cases that cover the identified lines, including both positive and negative test cases. Users can specify the number of lines to be covered by the generated tests. Then, for each identified line, PySTAAR generates corresponding positive and negative tests.

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

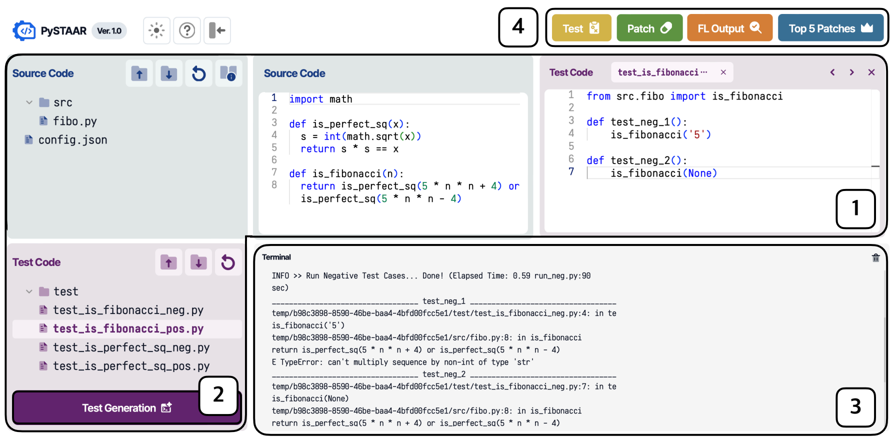

Figure 2: Web interface of PySTAAR. Initially, the 'Test' and 'Patch' buttons are deactivated. 'Test' is activated after tests are generated. 'Patch' is activated after pressing the Test button. 'FL Output' is activated after pressing 'Patch'.

errors [10]. 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 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 them against the test cases and checking whether the type errors are resolved. Patches that successfully eliminate the errors are considered plausible; otherwise, they are discarded. PySTAAR further checks whether each plausible patch preserves the original function's return type. Only patches that maintain the original return type are considered valid and are provided as the final output of the framework.

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 path of the Python file to be analyzed, `source_directory` refers to the root directory of the source code, and `project_directory` refers to the directory where the project is located. 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.

Table 1: Evaluation on 7 samples from CrowdQuake [6]. #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	✓

3 Web Interface

To assist novice developers, PySTAAR also provides a web-based interface designed to facilitate user interaction and provide comprehensive, end-to-end visualization of the entire process. Figure 2 presents the four main components of this interface: (1) **Code Editor**, which displays the source and test files for viewing and editing. Users can upload code directly, and upon uploading a Python project, `config.json` and `requirements.txt` are automatically generated (pre-filled but editable), with detailed guidance provided on the website. (2) **Test Generation Panel**, where clicking the *test generation* button produces test cases for the selected source code, with options such as the number of test cases and the target function. (3) **Terminal Output**, which shows real-time results of test execution and patch generation. (4) **Control Toolbar**, offering commands to run all tests, trigger automated patch generation,

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

display fault localization output, and present the top five patch candidates.

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 [6], 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 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 3 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 fixing known type errors in open-source Python projects. From the TypeBugs benchmark [10], we selected three projects of varying sizes (5.8–519.6kLoC): `requests`[3], `pandas`[4], and `salt` [5]. For each project, we used the most recent type error included in TypeBugs 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. Detailed information about the bugs and the generated patches is provided in our repository.

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

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.

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] facebook. 2017. Pyre-check: Performant type-checking for python. https://github.com/facebook/pyre-check
[3] github. 2024. Octoverse: AI leads Python to top language as the number of global developers surges. https://github.blog/news-insights/octoverse/octoverse-2024/
[4] google. 2015. Pytype: A static type analyzer for Python code. https://github.com/google/pytype
[5] 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*.
[6] 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*.
[7] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for Python. In *ACM/IEEE International Conference on Software Engineering: Companion Proceedings*.
[8] 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.
[9] microsoft. 2019. Pyright: Static Type Checker for Python. https://github.com/microsoft/pyright
[10] 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*.
[11] Wonseok Oh and Hakjoo Oh. 2024. Towards Effective Static Type-Error Detection for Python. In *IEEE/ACM International Conference on Automated Software Engineering*.
[12] OpenAI. 2025. GPT-4o. https://platform.openai.com/docs/models/gpt-4o.
[13] 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*.
[14] python. 2012. Mypy: Static Typing for Python. https://github.com/python/mypy
[15] 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*.
[16] Ruofan Yang, Xianghua Xu, and Ran Wang. 2025. LLM-enhanced evolutionary test generation for untyped languages. *Automated Software Engineering* 32, 1 (2025), 20.