# FixKit: A Program Repair Collection for Python

Marius Smytzek
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
marius.smytzek@cispa.de

Martin Eberlein
Humboldt-Universität zu Berlin
Berlin, Germany
martin.eberlein@hu-berlin.de

Kai Werk
Humboldt-Universität zu Berlin
Berlin, Germany
werkkai@hu-berlin.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@hu-berlin.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

## Abstract

In recent years automatic program repair has gained much attention in the research community. Generally, program repair approaches consider a faulty program and a test suite that captures the program's intended behavior. The goal is to automatically generate a patch that corrects the fault by identifying the faulty code locations, suggesting a candidate fix, and validating it against the provided tests. However, most existing program repair tools focus on Java or C programs, while Python, one of the most popular programming languages, lacks approaches that work on it.

We present FixKit a collection of *five* program repair approaches for Python programs. Moreover, our framework allows for easy integration of new repair approaches and swapping individual components, for instance, the used fault localization. Our framework enables researchers to effortlessly compare and investigate various repair, fault localization, and validation approaches on a common set of techniques.

## CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**; *Genetic programming*; **Software libraries and repositories**.

## Keywords

Python, Program Repair

**ACM Reference Format:**
Marius Smytzek, Martin Eberlein, Kai Werk, Lars Grunske, and Andreas Zeller. 2024. FixKit: A Program Repair Collection for Python. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Finding and fixing bugs in software is a time-consuming and error-prone task. Especially in large software systems, it is challenging to identify the root cause of a bug and to provide a fix that does not introduce new bugs. Moreover, the fix should generalize to the actual purpose of the program.

To address these challenges, researchers have developed automatic program repair approaches that aim to automatically generate patches for faulty programs. The goal is to identify the faulty code locations, suggest a candidate fix, and validate it against the provided test suite. In recent years, program repair has gained much attention in the research community, with a variety of approaches focusing on different aspects of the repair process.

However, the majority of existing program repair tools focus on Java or C programs [3, 9], while Python as one of the most popular programming languages lacks approaches that work on it. Since Python is widely used in various domains, including web development, data science, and machine learning, it is crucial to consider program repair that works on Python programs. In this paper, we present FixKit, a collection of five program repair approaches for Python programs. Moreover, our framework allows for easy integration of new repair approaches and swapping individual components, for instance, the used fault localization.

Our framework enables researchers to effortlessly compare and investigate the aspects and techniques of various repair, fault localization, and validation approaches on a common base.

To summarize, our contributions are as follows:

**Program Repair Collection** We provide a collection of *five* known program repair approaches for Python programs.

**Common Framework** We provide a common framework that allows for easy integration of new repair approaches and swapping individual components.

**Subjects** We provide an integration of Tests4Py in our framework, a set of subjects based on the faults in BugsInPy to evaluate the repair approaches and their components.

The remainder of this paper is structured as follows: In Section 2, we provide an overview of program repair and the various approaches and techniques comprised by FixKit. In Section 3, we describe some implementation details of FixKit. In Section 4, we provide instructions on how to install and use FixKit. In Section 5, we present a preliminary evaluation to verify the functional correctness of the included repair approaches. In Section 6, we discuss potential threats to the validity of the presented work. In Section 7, we discuss some related work. Finally, in Section 8, we conclude and provide an outlook on future work that we will conduct with the help of FixKit.

## 2 Program Repair

As mentioned in the introduction, program repair aims to automatically generate patches for faulty programs. Generally, we can distinguish between two approaches to program repair: *generate-and-validate* and *semantic-based* program repair. The *generate-and-validate* approaches generate a set of candidate patches and validate them against the provided test suite. Such approaches are often based on search-based techniques, for instance, genetic programming or evolutionary algorithms to iteratively improve the candidate patches by considering the provided test suite to produce a fitness value for each candidate produced during the generation. In contrast, the *semantic-based* approach uses the semantics of the program to generate a patch, for instance, by using symbolic execution or constraint solving to infer a correct code snippet.

In this paper, we focus on the *generate-and-validate* approach, as it comprises not only the more widespread approaches in the field of program repair but also more general approaches not focusing on only a small subset of faults to repair.

These approaches generally consist of three main components: *fault localization*, *patch generation*, and *patch validation.*

**Fault Localization** identifies the faulty code locations in the program based on the provided test suite. The general idea is that a code location is considered more likely to be faulty if it is executed more often when a test fails and less often when a test passes.

**Patch Generation** suggests a candidate fix for the identified faulty locations. The candidate fix is usually generated by applying mutations to the faulty code locations.

**Patch Validation** validates the candidate fix against the provided test suite. The candidate fix is considered the correct patch if it passes all tests.

While *fault localization* is usually embedded in the repair algorithm once, *patch generation* and *patch validation* are often interleaving components that are iteratively applied to improve the candidate patches.

### 2.1 Fault Localization

Fault localization is the process of identifying the faulty code locations in a program based on the provided test suite. The goal is to identify the code locations that are most likely to be the cause of the failing tests. Usually, fault localization techniques leveraged by program repair approaches are based on statistical metrics, that favor code locations, commonly lines, that are executed more often when a test fails and less often when a test passes. Common metrics are TARANTULA, OCHIAI, and JACCARD.

FixKit provides a set of fault localization techniques that can be easily swapped and compared, which do not only include the investigation of lines as code locations but various other granularities and aspects, for instance, conditions or branches.

### 2.2 Approaches

FixKit includes a set of five program repair approaches that can be easily compared and investigated. Those approaches comprise GENPROG [9], MUTREPAIR [1], KALI [5], CARDUMEN [4], and AE [8].

*GENPROG.* GENPROG [9] employs genetic programming and the competent programmer's hypothesis to generate patches for faulty programs. Genetic programming is used to evolve a population of patches that are applied to the faulty program. The patches are evaluated based on the provided test suite and a fitness function. The fitness function is based on the number of failing tests that are fixed by the patch and the number of passing tests that are still passing with the patch. The fitness is defined as

$$\begin{aligned} \mathrm{F}_{\mathrm{GenProg}}(P) = w_{PosT} &\times |\{t \in PosT | P \text{ passes } t\}| \\ +w_{NegT} &\times |\{t \in NegT | P \text{ passes } t\}| \end{aligned} \tag{1}$$

where $P$ is the patch, $PosT$ is the set of passing tests, $NegT$ is the set of failing tests, and $w_{PosT}$ and $w_{NegT}$ are the weights for the passing and failing tests, respectively. For each generation, the genetic programming selects the best candidates based on the fitness function and applies random deletions, insertions, or replacements of statements to iteratively find a correct patch. These mutations target those statements that are most likely to be the cause of the failing tests based on the statistical fault localization.

The competent programmer's hypothesis states that a developer is likely to already implemented an almost correct source code and the fix is only a small change away. In this context, GENPROG does not infer a new statement to insert or replace an existing one but rather selects a statement from the faulty program itself.

*MUTREPAIR.* In contrast to GENPROG, MUTREPAIR [1] leverages mutation operators to transform suspicious if conditions. The approach considers the if conditions as the most likely cause of the failing tests based on the fault localization and applies mutations that transform the operators in these conditions, for instance, from `==` to `!=` or `<` to `<=`.

Moreover, MUTREPAIR performs an exhaustive search on the current population, i.e., it considers all candidates in the current population and applies all mutation operators to each of them, in contrast to GENPROG, which applies selected mutations to randomly selected candidates.

*KALI.* The KALI [5] approach was designed to identify inadequat test set. However, it also works as a program repair approach following the GENPROG repair loop but leverages mutation operators that remove or skip certain parts of the code that are considered the most likely cause of the failing tests based on the fault localization. The approach considers the deletion of statements as the primary mutation operator. Moreover, KALI introduces mutation operators that skip certain parts of the code by setting if conditions to `True` or `False` or by inserting return statements such that a function exits early. Similar to MUTREPAIR, KALI performs an exhaustive search on the current population and applies all mutation operators to all candidates in the population.

*CARDUMEN.* In contrast to the other approaches, CARDUMEN [4] is specifically designed to produce as many patches as possible by considering an immersive search space. The approach proposes the creation of templates based on the statements in a program. These templates introduce placeholders for the variables in the statements. When selected for mutation, the placeholders are replaced by arbitrary variables present in the scope of the statement to repair.

CARDUMEN considers a probabilistic model to select the variables to replace the placeholders. The model prioritizes variable combinations that are more likely to be used in a statement together, i.e. the more often a variable combination is used in the program, the more likely it is to be selected to replace the placeholders.

*AE.* The AE [8] approach builds on GENPROG and eliminates the randomness in the patch generation process by considering all mutations up to a certain degree $k$. Moreover, the approach only considers candidates that are unique in the sense that a candidate is not equivalent to any candidate already verified.

It is worth highlighting that AE does not leverage evolutionary algorithms and a fitness function to identify the best candidates, but rather considers a candidate as a fix if and only if it passes all tests. Moreover, in contrast to GENPROG, AE is completely deterministic and does not consider random mutations but rather considers all possible mutations it can apply.

To repair a program, AE leverages lazy evaluation to generate a set of candidates with all possible mutations up to a degree $k$, i.e., candidates that are generated by applying one mutation, two mutations, and so on up to $k$ mutations. In contrast to GENPROG, AE only uses deletions and insertions as mutations, since replacements are covered by the first deleting a statement and then inserting another one, or vice versa. AE then leverages a repair strategy to select the best candidate from the set of candidates based on the fault localization information.

Next AE verifies that the selected candidate is unique by checking if it is equivalent to any other candidate that has already been verified. It considers syntactic equivalence, dead code, and the order of statements to determine if two candidates are equivalent. If the candidate is unique, the candidate gets validated against the provided test suite by sequentially running all tests until the candidate passes all tests or it is discarded as soon as one test fails. To determine the order of tests, AE employs a test strategy. The test strategy prioritizes tests that are more likely to fail based on previous executions of the tests.

As soon as a candidate passes all tests, it is considered as the fix for the fault and the repair terminates, otherwise, the next candidate is selected until all candidates have been verified.

## 3 Implementation

FixKit leverages two approaches for the fault localization. It allows a fast localization based on the Python coverage module that extracts the needed values for the metrics from the collected per-test coverage information. Moreover, it provides a more fine-grained and powerful fault localization by embedding SFLKit [7], a fault localization framework that allows for various fault localization techniques and granularities.

The patch generation and validation are implemented for each repair approach individually. However, FixKit provides a common interface to swap individual components, for instance, the leveraged fault localization technique or the fitness function. Moreover, it allows for easy integration of new repair approaches.

```python
repair = PyGenProg.from_source(
  src= "subjects/middle",
  excludes=["tests.py"],
  localization=CoverageLocalization(
    "subjects/middle",
    cov="middle",
    metric="Ochiai",
    tests=["tests.py"],
  ),
  population_size=40,
  max_generations=10,
)
patches = repair.repair()
```

**Figure 1: Example usage of FixKit to repair a subject located in `subjects/middle` with GENPROG.**

Additionally, we integrated Tests4Py [6], a benchmark of faulty programs and their corresponding test suites based on the faults in BugsInPy [11], into the framework. This allows for easy evaluation of the repair approaches on a common set of subjects.

Moreover, FixKit allows for parallel execution of the repair candidates for the validation step to speed up the repair process.

## 4 Usage

Installing FixKit is as simple as running `pip install fixkit`.[1] After installing the package, FixKit provides a library of all techniques and repair approaches that can be leveraged in a Python program. Figure 1 shows an example usage of FixKit to repair a subject located in `subjects/middle` with GENPROG. The resulting patches if any are returned by the `repair` method of any repair approach.

## 5 Preliminary Evaluation

We thoroughly tested each component of FixKit to ensure that the repair approaches, fault localization techniques, and validation strategies work as expected by providing a set of unit tests that verify the outcome of each component with assertions based on what we expect.

Moreover, we conducted a preliminary evaluation of the repair approaches on the infamous middle program. We leveraged Tests4Py [6] to access this middle program, which Tests4Py assigns the identifier middle$_2$. Forwardly, we verified the functional correctness of all included approaches, the fault localization techniques, and the validation strategies.

We manually investigated each execution by stepping through the repair process and verifying that our implementations stayed true to the described approaches in their respective sources. Moreover, we verified that the fault localization techniques and the

[1] For the latest version, please refer to the GitHub project.

**Table 1: Results of the preliminary evaluation.**

| Subject | GENPROG | MUTREPAIR | KALI | CARDUMEN | AE |
|---|---|---|---|---|---|
| middle | ✓ | ✗ | ✗ | ✓ | ✓ |

validation strategies worked as expected by checking the generated patches and the validation results, which are shown in Table 1.

These results show that neither MUTREPAIR nor KALI were able to generate a correct patch for the middle program. In contrast, GENPROG, CARDUMEN, and AE were able to generate a correct patch for the middle program. MUTREPAIR and KALI failed to generate a correct patch for the middle program because of their set of mutation operators that did not cover a possible fix for the fault in the middle program.

## 6 Threats to Validity

Even though we thoroughly investigated the implemented repair approaches, fault localization techniques, and validation strategies and tried to ensure that we stayed true to their respective sources, there might be errors in our implementation such that we did not implement the repair algorithms as described in the source. However, by manually stepping through each algorithm we are convinced that our implementations do not contain any major flaws and the various techniques work as expected.

Moreover, the preliminary evaluation was conducted on a single subject, the middle program. While this subject is well-known in the field of program repair, it might not be representative of the general performance of the repair approaches. However, since we want to verify the functional correctness of the repair approaches, we are convinced that the results of the preliminary evaluation are valid.

## 7 Related Work

Our benchmark is closely related to ASTOR [3], an automatic software repair framework in Java. Since its initial publication, the framework has been continuously expanded with other automatic program repair approaches [4, 10]. These expansions have integrated advanced techniques such as deep learning models for identifying and fixing bugs (as seen in DeepRepair [10]) and specialized patch generation methods (as demonstrated by Cardumen [4]). The ASTOR framework's evolution highlights the importance of combining multiple strategies to enhance the robustness and effectiveness of automated software repair.

Similarly, FixKit is a Python library that functions as a collection of automated repair tools. Like ASTOR, FixKit leverages various methodologies to address a wide range of bugs in Python programs. The modular design of FixKit allows for the integration of new repair techniques as they are developed, ensuring that the library remains at the forefront of automated software repair technology. This adaptability is crucial for maintaining relevance in the rapidly evolving field of software development, where new bug patterns and programming paradigms continuously emerge.

Furthermore, while ASTOR is tailored specifically for Java, FixKit exemplifies a move towards more versatile tools that can potentially be extended to support multiple programming languages. This trend underscores the necessity for repair tools that are not only powerful but also adaptable to diverse coding environments and practices.

## 8 Conclusion and Future Work

In this paper, we presented FixKit, a collection of five program repair approaches for Python programs. Moreover, we provided a common framework that allows for easy integration of new repair approaches and swapping individual components, for instance, the used fault localization.

As part of our future work, we plan to extend the set of repair approaches and fault localization techniques, especially by integrating diagnostic techniques that help to identify the root cause of the fault, for instance, AVICENNA [2]. Moreover, we plan to conduct a thorough empirical study to investigate the effectiveness of the repair approaches on a larger set of subjects in Python to provide insights into the strengths and weaknesses of the repair approaches.

Our framework is available as an open-source project on GitHub:

https://github.com/smythi93/fixkit

## Acknowledgments

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – GR 3634/4-2 Emperor (261444241).

## References

[1] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74, 2010.

[2] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 438–449, New York, NY, USA, 2023. Association for Computing Machinery.

[3] M. Martinez and M. Monperrus. Astor: a program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 441–444, New York, NY, USA, 2016. Association for Computing Machinery.

[4] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In T. E. Colanzi and P. McMinn, editors, *Search-Based Software Engineering*, page 65–86, Cham, 2018. Springer International Publishing.

[5] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[6] M. Smytzek, M. Eberlein, B. Serce, L. Grunske, and A. Zeller. Tests4py: A benchmark for system testing, 2024.

[7] M. Smytzek and A. Zeller. Sflkit: a workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1701–1705, New York, NY, USA, 2022. Association for Computing Machinery.

[8] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366, 2013.

[9] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 364–374, 2009.

[10] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, 2019.

[11] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1556–1560, New York, NY, USA, 2020. Association for Computing Machinery.