Empirical Software Engineering (2022) 27: 48
https://doi.org/10.1007/s10664-021-10087-1

# Mining Python fix patterns via analyzing fine-grained source code changes

**Yilin Yang[1] · Tianxing He[1] · Yang Feng[1] · Shaoying Liu[2] · Baowen Xu[1]**

Accepted: 8 November 2021 / Published online: 28 January 2022


**Abstract**

Many code changes are inherently repetitive, and researchers employ repetitiveness of the code changes to generate bug fix patterns. Automatic Program Repair (APR) can automatically detect and fix bugs, thus helping developers to improve the quality of software products. As a critical component of APR, software bug fix patterns have been revealed by existing studies to be very effective in detecting and fixing bugs in different programming languages (e.g., Java/C++); yet the fix patterns proposed by these studies can not be directly applied to improve Python programs because of syntactic incompatibilities and lack of analysis of dynamic features. In this paper, we proposed a mining approach to identify fix patterns of Python programs by extracting fine-grained bug-fixing code changes. We first collected bug reports from GitHub repository and employed the abstract syntax tree edit distance to cluster similar bug-fixing code changes to generate fix patterns. We then evaluated the effectiveness of these fix patterns by applying them to single-hunk bugs in two benchmarks (BugsInPy and QuixBugs). The results show that 13 out of 101 real bugs can be fixed without human intervention; that is, the generated bug patch is identical or semantically equivalent with developer's patches. Also, we evaluated the fix patterns in the wild. For each complex bug, 15% of the bug code could be fixed, and 37% of the bug code could be matched by fix patterns.

**Keywords** Pattern mining · Fix pattern · Program repair · Bug fix changes

## 1 Introduction

Python design philosophy emphasizes code readability and flexibility, making it widely applicable in various development fields and especially popular with novices (Xu et al. 2016; Chen et al. 2018; Hu and Zhang 2020). Python dynamic features support reflection,

Communicated by: Shaowei Wang, Tse-Hsun (Peter) Chen, Sebastian Baltes, Ivano Malavolta, Christoph Treude, Alexander Serebrenik

This article belongs to the Topical Collection: *Collective Knowledge in Software Engineering*

✉ Yang Feng
fengyang@nju.edu.cn

Extended author information available on the last page of the article.

allowing its programs to access their types and structures at runtime, and to dynamically modify their execution status (Van Rossum and Drake 1995; Sanner et al. 1999; Åkerblom et al. 2014). However, these dynamic features also become a resistance in terms of tool support development (Monat et al. 2020). Åkerblom et al. (2014) proved that real-world Python programs leverage built-in dynamic features to exhibit behavior that is inherently hard to type. Wang et al. (2015) and Chen et al. (2018) have shown that the code changes of dynamic features are related to bug-fixing activities significantly. Hence, it is essential to understand Python dynamic features and reveal potential programming defects, as they help developers to understand bug characteristics and thus improve coding quality.

In this paper, we aim to discover fix patterns in Python programs and help developers to understand common and specific bug fixing in Python programs. Mining fix pattern is crucial and indispensable for implementing the fix pattern-based Automated Program Repair (APR) tool. Many code changes are inherently repetitive (Jiang and Su 2009; Nguyen et al. 2013a; Negara et al. 2014; Hindle et al. 2016), and researchers employed the repetitiveness to identify intrinsic features of programs that implement automated techniques such as fix pattern-based APR (Pan et al. 2009; Kim et al. 2013; Hanam et al. 2016; Liu et al. 2018; Cotroneo et al. 2019). Pan et al. (2009) proposed Java bug fix patterns to analyze how developers fix bugs. Kim et al. (2013) manually summarized 10 fix patterns from human-written patches. Hanam et al. (2016) conducted a comprehensive study of popular vulnerability patterns in server-side JavaScript code. Liu et al. (2018) employed CNN techniques to extract features from the bug patches to mine fix patterns. Cotroneo et al. (2019) conducted an empirical study on the repetitive pattern of vulnerability remediation changes in three OpenStack projects.

Unfortunately, the existing fix pattern-based APR tools can not be applied directly to Python programs because Python syntax is incompatible with that of Java, and existing researches have failed to systematically analyze Python dynamic features. Furthermore, as existing fix pattern mining methods, data pre-processing (i.e., code differencing) heavily relies on AST differencing tools such as GumTree (Hanam et al. 2016; Liu et al. 2018) and ChangeDistiller (Fluri et al. 2008; Wang et al. 2018); this may lead to the limited applicability of these research methods when certain programming languages do not have corresponding advanced AST differencing tools. Therefore, we attempted to propose an approach that can automatically mine fix patterns based on historical bug-fix information. This fix pattern mining approach can be applied not only to Python but also to other programming languages.

Recent studies (Saha et al. 2019; Noda et al. 2020) have shown that most state-of-the-art APR tools are designed to focus on single statement bugs, and though some APR tools can fix multi-hunk bugs (Mechtaev et al. 2016; Saha et al. 2019), the number (classes) of such fixable multi-hunk bugs is limited. Therefore, in this study, we focused on summarizing fix patterns for repairing single-hunk bugs. We restricted the bug patch to a single contiguous chunk of code.

The goals of this paper are twofold, namely (1) to propose a systematic and automated mining approach for discovering pervasive fix patterns and (2) to identify frequently occurring bug fix patterns in cross-project Python programs.

**Research Question** We present our study, which focuses on the following four research questions (RQs):

– **RQ1.** ***What are the common fix patterns in Python?***

We collected bug-fixing code changes frequently occurring in open-source software and identified common fix patterns. We described the common fix patterns in Section 4.

- **RQ2.** ***Which fix patterns are specific to Python?***
  Besides common fix patterns, we also analyzed the Python dynamic features and summarized that the fix patterns are specific to Python. We described the Python fix patterns in Section 5.
- **RQ3.** ***How many single-hunk bugs from BugsInPy and QuixBugs can be fixed by fix patterns?***
  We selected single-hunk bugs from the two benchmarks and applied fix patterns to repair these bugs. We evaluated the effectiveness of fix patterns in fixing single-hunk bugs in Section 6.
- **RQ4.** ***Are the fix patterns we proposed effective in practice?***
  We collected bug reports from four open-source projects as a test set, and then we defined and selected complex bugs from the test set. We evaluated the potential of fix patterns to fix the complex bugs in Section 6.

**Contribution** This paper makes the following primary contributions:

- We propose a novel technique for automatically mining fix patterns based on fine-grained bug-fixing code changes.
- We propose 29 fix patterns, which are 11 common fix patterns and 18 Python special patterns. We evaluate the effectiveness of the identified fix patterns by applying them to address real-world bugs.
- We conduct a comprehensive study of pervasive bug fix patterns in a large number of repositories.[1]

**Paper Organization** The remainder of this paper is organized as follows.

We describe the background in Section 2, which introduces the Python dynamic features and Fix Pattern-based Automated Program Repair. We present the methodology in Section 3, defining the basic terminologies applied in this study and describing the methodology steps. In Sections 4 and 5, we present the common fix patterns and Python fix patterns, whose descriptions and examples are both provided. Then, we describe the usage of fix patterns and evaluate the effectiveness of these fix patterns in Section 6. After that, we discuss threats to the validity of this study in Section 7 and related work in Section 8. Finally, we conclude with future work in Section 9.

## 2 Background

### 2.1 Python Dynamic Features

Python is a typical dynamically typed language that does not enforce to check type safety at compile time (static checking) but defers such checks to run time (dynamic checking), reducing development costs and providing the flexibility required by specific areas such as data processing (Tratt 2009). Holkner and Harland (2009) categorized the Python dynamic features into four sets: *reflection, dynamic typing, dynamic objects, and dynamic code*. Table 1 shows the built-in dynamic feature functions of Python

[1] https://github.com/SATE-Lab/PyFPattern

**Table 1** Bulid-in function (Python 3.8.X) for python dynamic feature

| Categories | Function | | | | |
|---|---|---|---|---|---|
| Introspection | any() | all() | callable() | dir() | getattr() |
| | hasattr() | isinstance() | issubclass() | globals() | locals() |
| | type() | vars() | | | |
| Self-modification | delattr() | setattr() | | | |
| Dynamic Code | compile() | eval() | exec() | | |

**Reflection** The programming language that supports reflection allows its programs to have run time access to their types and structure and to be able to modify their behavior dynamically. Tratt (2009) divides reflection into three activities, namely

- *Introspection.* Every object in Python has attributes and methods. Through introspection, we can dynamically inspect Python objects. Code introspection examines classes, methods, objects, modules, keywords and gets information about them. Introspection reveals useful information about program objects.
- *Self-modification.* Python object can change its own structure. For example, we can dynamically add an attribute through $setattr()$.
- *Intercession.* Python object can change its own behavior. As shown in Fig. 1- Intercession, it can return the new behavior by overriding the $\_\_getattribute\_\_()$.

**Dynamic Typing** Variables in Python are neither statically declared nor typed. Any variables can hold any type, and the type of the value held may change during program execution (Holkner and Harland 2009; Monat et al. 2020). As shown in Fig. 1-Dynamic Typing, parameter $x$ is a reference to object $a$. With an immutable object, change $x$ does not affect $a$, while with mutable objects, such as $b$, changing $x$ may change the original object.

**Dynamic Object** Python classes can be constructed before calling at run time (Holkner and Harland 2009). In Python standard library, classes can be dynamically created using the *X = type('X', (object,), dict(a=1))*, as shown in Fig. 2-Dynamic Object.

*Intercession*

```
class Test (object) :
    x = 100

    def __getattribute__ ( self , name ) :
        if  name.startswith ( "get_" ) :
            v = object . __getattribute__ ( self , name[4:] )
            return lambda : v
        else:
            return object . __getattribute__ ( self, name )

if __name__ == '__main__':
  i = Test ( )
  print( i.x )   >>100
  print( i.get_x( ) )  >>100
```

*Dynamic Typing*

```
def Test ( x , y ) :
    x = '***'
    y [ 0 ] = 100
    print ( x , y )  >> ***, [100,2,3]


a = 1
b = [1,2,3]

Test ( a , b )
print ( a , b )  >>1, [100,2,3]
```

**Fig. 1** Intercession & dynamic typing example code

*Dynamic Object*

```
def func(self):
  print('Hello, world!')

Hello = type('Hello', (object,), dict(hello=func))
h = Hello()
h.hello()

print(type(Hello)) # class type is <class 'type'>   >> <class 'type'>

print(type(h)) # instance type is class name   >> <class '__main__.Hello'>
```

*Dynamic Code*

```
def func(str):
  s = eval(str)
  return s

str1 = func('1+1')
print(str1)    >> 2

str1 = func("'*' * 5")
print(str1)   >> ****
```

**Fig. 2** Dynamic object & dynamic code example code

**Dynamic Code** Python can construct code at run time from source code (Holkner and Harland 2009). For example, we can use $eval()$ to evaluate arbitrary string expressions as Python code at run time, as shown in Fig. 2-Dynamic Code.

### 2.2 Fix Pattern-based Automated Program Repair

Fix pattern is also termed as fix template (Liu and Zhong 2018) or program transform scheme (Hua et al. 2018), usually obtained from manual summarization, pre-definition, statistics and mining (Liu et al. 2019b).

Summarizing fix patterns is challenging because these patterns are obtained from various heterogeneous source codes, such as across application domains and software project versions. Kim et al. (2013) manually summarized 10 fix patterns from human-written patches, which were collected from Eclipse JDT. Durieux et al. (2017) proposed NPEfix, an APR tool based on 9 fix patterns specific to null pointer exceptions to detect null pointer defects. Hua et al. (2018) proposed SketchFix, an APR tool with 6 pre-defined fix patterns. Liu et al. (2018) employed CNN techniques to extract features from the bug patches to mine fix patterns. Subsequently, Liu et al. (2019a) proposed AVATAR, an APR tool that applies the generated fix patterns to fix semantic bugs.

Software systems inevitably appear with various defects, the limited human resources are not sufficient to fix all known bugs (Habib and Pradel 2018). Researchers have proposed various automation techniques to address this challenge. In existing studies, researchers generally exploit the repetitiveness of code changes to identify the intrinsic program features and thus accomplish some specific tasks, such as fix pattern-based automated program repair (APR). The basic principle of fix pattern-based APR is to abstract code changes into a pattern and apply the contextual information of the abstract syntax tree node of the buggy code to match the contextual constraints (Liu et al. 2019a; 2019b), as shown in Fig. 3.

Existing studies have revealed several fix patterns for various programming languages, e.g., Java; however, these studies can not be directly applied to Python programs because of syntactic incompatibilities and lack of analysis of dynamic features. These situations motivate us to expand the existing literature by enhancing our understanding of Python language features and fix patterns.

## 3 Research Methodology

We defined the basic terminologies applied in this study with reference to the existing literature (Section 3.1). Our methodology includes the following steps, as summarized in Fig. 4.

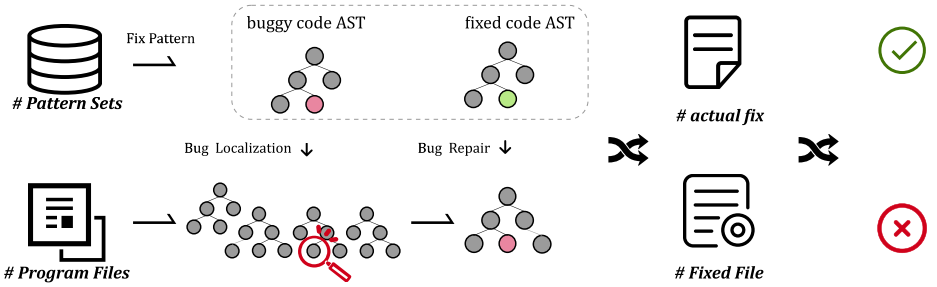



**Fig. 3** Fix pattern-based automated program repair

First, we collected bug reports from the GitHub repository (Section 3.2); Second, we mined and normalized the single-hunk bugs (Section 3.3); Third, we clustered these hunks according to their Abstract Syntax Tree (AST) edit distance (Section 3.4); Finally, we manually reviewed the cluster results (Section 3.5).

### 3.1 Terminology

The terminologies applied in this study are defined as follows.

- **Bug Report.** We collected data sets from GitHub. This paper refers to the officially merged issues as bug reports, which contain bug symptom descriptions, bug patches, and commit information.[2]
- **Bug Patch.** Bug patch is a pair of bug-fixing code snippets, bug code from a buggy version, and fixing code from its updated version (Fluri et al. 2007; Negara et al. 2014; Liu et al. 2018).
- **Single-hunk.** Hunk is a contiguous set of bug-fixing code changes (Wen et al. 2016). Single-hunk bugs require program fixes at a single location or a set of contiguous locations (Saha et al. 2019).
- **Fix Pattern.** Fix pattern is a pair of a code context extracted from a buggy code block and a set of change operations (i.e. delete, add, move, change). Fix pattern can be applied to a given buggy code block to generate fixing code (Liu et al. 2018; Liu et al. 2019b).

According to recent researches (Nguyen et al. 2013a; Hanam et al. 2016), the APR methods should concentrate on the change fragments with small sizes of 2–6 lines. In this paper, we restricted the bug patch confined to a single contiguous chunk of code, i.e., less than consecutive 6 lines of the bug-fixing code changes.

### 3.2 Collecting Bug Report

As shown in Table 2, there are 36 popular Python projects from GitHub studied in this paper. These projects come from different areas, including machine learning, data processing, web, media, development, and deployment. This data set was collected in January 2020. We selected these projects based on two criteria (Chen et al. 2018): one is that the project should maintain a relatively long traceable record on GitHub; the other is that the issue information

[2] https://guides.github.com/features/issues/

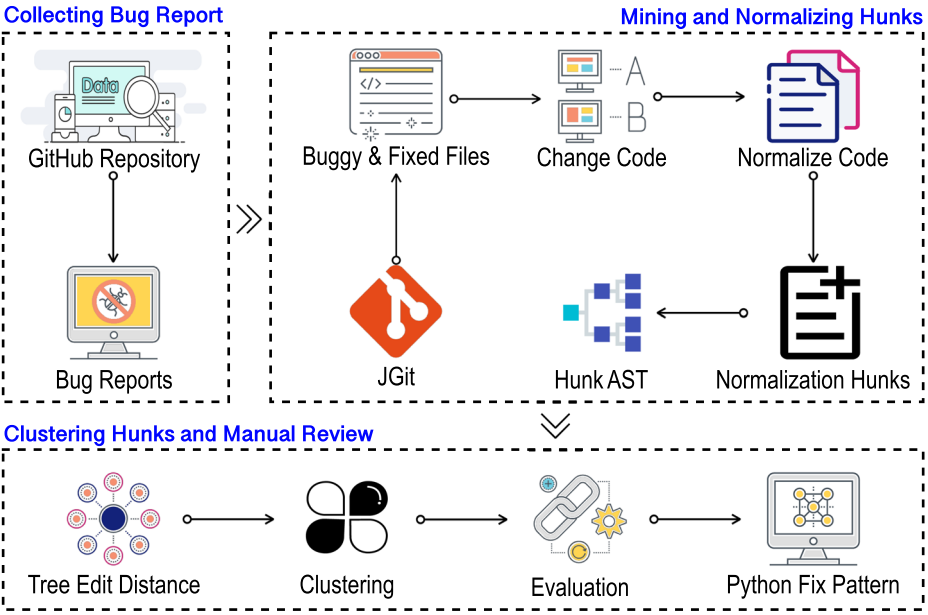



**Fig. 4** Overview of our fix patterns mining method

record of the project follows a certain template, and we can automatically extract bug reports by identifying the textual information.

We first collected all issues for these repositories via GitHub API. To obtain bug reports, we only focused on the issues that merged into the master branch. This process was to identify issues that are officially recognized to avoid invalid discussions. Then, we applied the approach of Vasilescu et al. (2015) to filter the bugs in the data set. We set up a list of keywords (Zhang et al. 2018), including *'bug', 'fix', 'wrong', 'error', 'nan', 'inf', 'issue', 'fault', 'fail', 'crash'*. Then we searched these keywords in issues' title, and labels. If any of the titles or labels of the issues contain at least one keyword, we identified it as a bug report. Finally, we obtained 64,011 bug reports. Here, we obtained many bug reports from the ansible project. We believe that it is related to the popularity and maintenance of the project community.

### 3.3 Mining and Normalizing Single-hunk Bugs

After collecting bug reports, we obtained the bug-fixing file changes for each bug report by *jGit* tool. We mined the single-hunk bugs and normalized these bugs by the method of Higo et al. (2020). The detailed process is as follows.

**Mining Single-hunk** After collecting bug reports, we can obtain each bug report's commit id, i.e., `sha` which is a unique identifier created as a new commit is recorded. As shown in Fig. 5, we first employed the *jGit* tool[3] to obtain all changed files for each bug report

[3]https://www.eclipse.org/jgit/

**Table 2** Descriptive information about target projects

| Domain | Repositories | SLOC(py) | Commits | PRs | T-Bugs | F-Bugs | M-Bugs |
|---|---|---|---|---|---|---|---|
| Data Science | explosion/spaCy | 27392 | 7698 | 1259 | 456 | 58 | 15 |
| | matplotlib/matplotlib | 117500 | 32472 | 9470 | 2700 | 306 | 88 |
| | numpy/numpy | 118470 | 22083 | 6872 | 3347 | 221 | 85 |
| | onnx/onnx | 26641 | 1433 | 1416 | 370 | 17 | 6 |
| | RaRe-Technologies/gensim | 33868 | 3878 | 1304 | 394 | 106 | 18 |
| | scikit-image/scikit-image | 46418 | 11903 | 2586 | 902 | 100 | 31 |
| | scikit-learn/scikit-learn | 127649 | 25021 | 8305 | 2075 | 374 | 90 |
| | scipy/scipy | 171435 | 22500 | 4916 | 2215 | 240 | 107 |
| | statsmodels/statsmodels | 189071 | 12538 | 2467 | 959 | 69 | 17 |
| | sympy/sympy | 382265 | 40891 | 7931 | 1643 | 67 | 23 |
| Development | ansible/ansible | 979985 | 48802 | 38095 | 17061 | 4975 | 2318 |
| | apache/airflow | 141460 | 7796 | 6252 | 870 | 205 | 74 |
| | getsentry/sentry | 214136 | 27994 | 12137 | 4402 | 743 | 336 |
| | hyperledger/fabric | 3236 | 1066 | 428 | 31 | 2 | 1 |
| | ipython/ipython | 36312 | 24195 | 5708 | 1268 | 31 | 10 |
| | kivy/kivy | 50747 | 12034 | 2830 | 803 | 123 | 51 |
| | nicolargo/glances | 11137 | 3644 | 438 | 139 | 6 | 3 |
| | pygame/pygame | 15233 | 8156 | 701 | 240 | 18 | 1 |
| Machine | apache/incubator-mxnet | 277069 | 10526 | 8426 | 2356 | 340 | 120 |
| Learning | apple/coremltools | 81316 | 672 | 269 | 89 | 7 | 4 |
| | chainer/chainer | 135649 | 30455 | 6427 | 2306 | 561 | 90 |
| | dmlc/gluon-cv | 36948 | 645 | 597 | 160 | 77 | 23 |
| | keras-team/keras | 47158 | 5342 | 3740 | 839 | 366 | 159 |
| | PaddlePaddle/Paddle | 141506 | 26154 | 11406 | 3308 | 356 | 134 |
| | pandas-dev/pandas | 269573 | 21519 | 13257 | 3893 | 316 | 122 |
| | pytorch/pytorch | 401139 | 23442 | 18252 | 1263 | 322 | 98 |
| | tensorflow/tensorflow | 588367 | 75891 | 12268 | 2937 | 607 | 231 |
| | Theano/Theano | 129492 | 28094 | 3972 | 1403 | 63 | 18 |
| Others | beetbox/beets | 36460 | 9241 | 1217 | 250 | 18 | 3 |
| | django/django | 243678 | 27785 | 12024 | 3314 | 264 | 86 |
| | HelloZeroNet/ZeroNet | 27610 | 3780 | 453 | 99 | 15 | 6 |
| | mailpile/Mailpile | 46722 | 6265 | 580 | 150 | 6 | 1 |
| | scrapy/scrapy | 27392 | 7698 | 1875 | 391 | 20 | 9 |
| | wagtail/wagtail | 66687 | 9527 | 2702 | 337 | 22 | 5 |
| | ytdl-org/youtube-dl | 120298 | 17507 | 2955 | 498 | 193 | 101 |
| | zulip/zulip | 122524 | 34050 | 8254 | 543 | 66 | 30 |
| Total | | 5492543 | 652697 | 221789 | 64011 | 11280 | 4514 |

*Projects URL*: https://github.com/Repositories;

*T-Bugs*: Total number of bugs;

*F-Bugs*: Changed single file bugs;

*M-Bugs*: Changed single method() bugs;

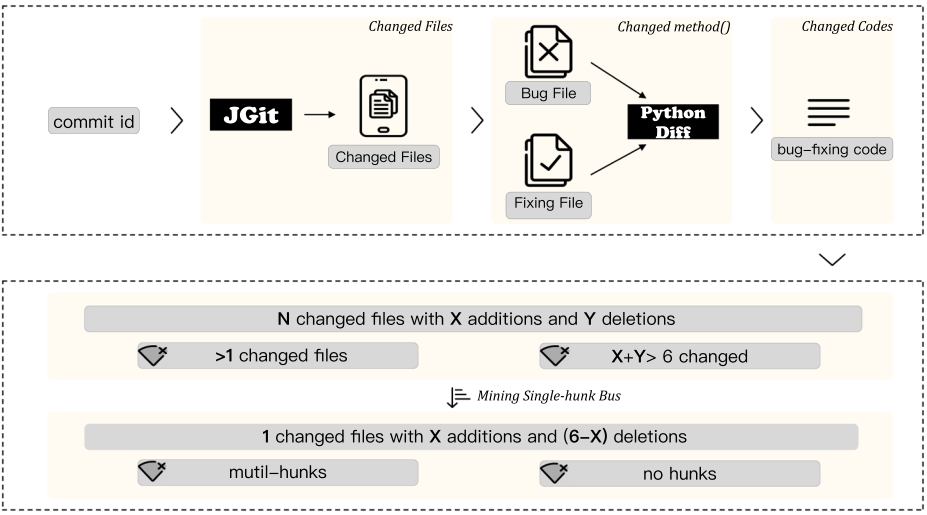



**Fig. 5** Mining single-hunk bugs

by its `sha`. We conducted this step on all bug reports and selected the commits which only have one changed file. Secondly, we applied the *Python Diff* tool[4] to compare *bug.py* and *fixing.py* to select the files that changed only one *method()*.

Thirdly, following the study of Nugroho et al. (2020), we used the histogram algorithm[5] to identify the code differences in bug-fixing code changes. The total number of bug codes and fixing codes should be less than 6, and the changed code should be consecutive.

**Normalizing Single-hunk** We normalized hunks at AST level. According to Higo et al. (2020), we normalized the variables to var#, where # means the number of variables in a single hunk. In each statement, the same number is assigned to the same name, and different numbers are assigned to different names. For example, `a=a+1` is normalized to `var0=var0+1`, but `a=b+1` should be normalized to `var0=var1+1`. We also normalized the arguments (arg#), numerical value (num#), and string (s). We did not normalize method names because the semantics of calling different Application Program Interface (APIs) are very different. For example, `cmd=self._build_command(self._play_context)` is normalized to `var0=var1._build_command(var1._play_context)`. Note that we believe that the build-in function names are special method names rather than API names in the ordinary sense.

In our technique, we only extracted modified bug code snippets for AST analysis. However, these snippets may be incomplete and cannot be transformed into AST. In this case, we need to complete the snippets so that they can be transformed into AST. For if statements, we only added 'pass' for if statements with incomplete semantics. For example, `if(x<1):` is formalize to `if(x<1):pass`

[4] https://github.com/petr-muller/pyff

[5] Histogram algorithm is the enhanced version of Patience. https://alfedenzo.livejournal.com

### 3.4 Clustering Single-hunk Bugs

To extract fix patterns from these normalized hunks, we first categorized the bug codes in single-hunks by AST types and number of lines, then we clustered each sub-category according to the bug-fixing code ASTs edit distance. The detailed process is as follows.

**Coarse-grained classification** We transformed the bug codes in hunks into AST and classified them according to their AST types. Then, we classified the above subclasses according to the number of bug code lines.

**Clustering bug-fixing code** We clustered each sub-category in above coarse-grained classification. We applied the APTED algorithm[6] (Pawlik and Augsten 2015; 2016) to calculate the AST edit distance to evaluate code similarity, whose working principle is to traverse the structure of the syntax tree and compare whether the structure and the node content are the same. Users can assign weights to node deletion, node insertion, and node renaming operations. Here, we set the weights of the delete, insert and rename node operations to one. To compare all the bug codes, we transformed the bug codes into ASTs, and calculated the edit distance of their ASTs. According to Chakraborty et al. (2018), we classified the bug code ASTs according to the edit distance from 0-5. As in the above step, we calculated the fixing code AST edit distance of all single-hunks and classified them according to the fixing code edit distance 0-5.

If the edit distance of the bug code and the edit distance of their corresponding fixing code is both 0, then the hunks are the same. We listed these hunks as *candidate hunks*. If both the edit distance of the bug code and the edit distance of their corresponding fixing code is 1-5, it means that these hunks are not same but similar. We listed these hunks as *suspicious hunks* and classified them by edit distance.

**Evaluation** We set two restrictions: (1) the number of cluster members is greater than 3; (2) bug-fixing code cannot come from duplicate code. For the first restriction, we believe that only recurring bug-fixing code changes can represent a type of bug fixes. For the second restriction, If we use duplicate bugs (i.e., bugs were found in duplicate code), the same bug may be counted repeatedly, which would lead to a faulty clustering result. For the clusters that met the criteria, we performed a manual review to ensure that they represented a type of bug fix.

### 3.5 Manual Review

Following existing researches (Cotroneo et al. 2019; Xia et al. 2019), the first three authors of this paper manually reviewed each cluster to assess whether the cluster actually represents a bug fix.

We first defined three hunk types reference to existing literature (Cotroneo et al. 2019), and then we adopted the open coding method (Seaman 1999) to qualitative analysis of each class. According to Cotroneo et al. (2019), we divided the clusters into three types:

*** bug-fix**, the changed code actually fixes the behavior of software, which can represent fixes for a bug type.

[6] https://github.com/JoaoFelipe/apted

* **fix-induced**, the changed code is a group of bug-fixing code changes but not represents an actual bug fix, e.g., adding new input parameters to a method, the method signature and method call must be changed corresponding.

* **refactoring**, the changed code does not modify the software behavior, e.g., better readability or encapsulation.

Then, we labeled all candidate hunks and suspected hunks with the above three types. After all the clusters were labeled, the authors argued for their disagreements and reached a consensus with reasonable results. Meanwhile, we invited one other author to participate in the discussion for the remaining few disagreements to reach a final consensus. We calculated Fleiss Kappa value (Fleiss 1971) to measure the agreement between the authors. Fleiss Kappa value can be divided into 5 intervals, which are slight [0.01, 0.20], fair (0.20, 0.40], moderate (0.40, 0.60], substantial (0.60, 0.80), and perfect (0.80, 1] (Shrout and Fleiss 1979; Hallgren 2012). Our Fleiss Kappa value is 0.72, which means that the agreement of the identified clusters is considered to be substantial.

Next, we eliminated the clusters labeled as **refactoring** because they do not represent a bug fix.

When the suspected hunks had the same semantics as the candidate hunks, we merged them and employed the candidate hunks as the pattern for these hunks. When the suspected hunks could not merge to any candidate hunks, we manually summarized them to a formal pattern. In addition, to further analyze the bug fix related to dynamic features in Python. We analyzed all bug-fixing code changes involving built-in dynamic feature functions (Table 1).

We termed the pattern for fixing common bug types (e.g., modifying if constraints, adding or removing arguments to method calls) (Pan et al. 2009) as **Common Fix Pattern**; the fixing scheme involving Python operators, dynamic features, and migration rules is termed **Python Fix Pattern**. As a result, 770 bugs matched the common fix pattern, and 244 bugs matched the Python fix pattern. Table 3 shows the distribution of bugs matched each fix pattern. Finally, we discussed the common fix pattern in section 4 and the Python fix pattern in section 5. Also, we summarized the Python migration pattern in Appendix.

## 4 Common Fix Pattern

### 4.1 Answer to RQ1: Common Fix Pattern

We summarized 11 common fix patterns by automatic mining and manual review, as shown in Table 4. We presented a catalog of fix patterns in Section 4.2-4.12 and provided the description and example for each pattern. Example code with a prefix "-" is the bug code, while code with prefix "+" is the fixing code.

**Discussion** Four patterns (C1, C3, C4, C7) are high frequency bug fixing schemes. It is because converting data types (C1), mutating method invocation expression (C2 - C4), and changing if statement constraints (C6 - C7) occur frequently and inevitably in any type of system. Also, we found some fix patterns (C5, C10, C11) specific to Python syntax. For example, Python *for* loop form as *'for <var> in <iterable>:'*, where iterable is a collection of objects, e.g., list. C5 pattern is employed to fix the data type of the iterable object. Because iterable is usually the return value of other expressions, the iterable data type often changes at runtime. For other instance, *Raise* and *With* are all the keywords not in Java. For usage of common fix pattern, the bug fixing methods can be divided into two types, one is to transform bug code only at the string level, e.g., C1 pattern; and the other is

**Table 3** The distribution of bugs matched each fix pattern

| Common Fix Pattern | | | | Python Fix Pattern | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | Quantity | No. | Quantity | No. | Quantity | No. | Quantity | No. | Quantity | No. | Quantity |
| C1 | 300 | C7 | 92 | P1 | 34 | P6 | 4 | P11 | 37 | P16 | 23 |
| C2 | 36 | C8 | 24 | P2 | 50 | P7 | 6 | P12 | 3 | P17 | 3 |
| C3 | 149 | C9 | 7 | P3 | 30 | P8 | 3 | P13 | 17 | P18 | 4 |
| C4 | 61 | C10 | 34 | P4 | 3 | P9 | 4 | P14 | 12 | | |
| C5 | 11 | C11 | 14 | P5 | 3 | P10 | 6 | P15 | 11 | | |
| C6 | 42 | | | | | | | | | | |

No. of patterns are from Tables 4 and 5 respectively

to traverse the entire bug file to collect information about variables, instances, arguments, etc. in the file as candidates for replacing the buggy objects. We described the usage of the fix pattern in Section 6.1.

### 4.2 (C1) Change of Assignment Expression

Fix Pattern:

```
C1.1:Change Data Type
-   obj = exp/var
+   obj = type(exp/var)

C1.2:Change Expression
-   obj = exp1
+   obj = exp2
```

**Description** Replacing the right-side expression content in the assignment statement. We divided this fix pattern into two sub-fix patterns, namely: (1) Converting the data type of the

**Table 4** Common fix patterns

| No. | Pattern |
|---|---|
| (C1) | Change of Assignment Expression |
| (C2) | Change of Method Call to a New Instance |
| (C3) | Method Call with Different Number of Arguments |
| (C4) | Method Call with Different Value of Arguments |
| (C5) | Change Loop Variable to Iterable Object |
| (C6) | Change of Precondition Check |
| (C7) | ADD/DEL Precondition Check |
| (C8) | Change of Exception Type. |
| (C9) | Change of Third-party Library Dependency |
| (C10) | Change of Error Messages Content |
| (C11) | Change of With Context Expression |

buggy object to another data type; (2) Replacing the buggy data with other objects (including variables, literals, instances) in the bug file. For instance, in example 1.1, *candidate* is converted to string type; in example 1.2, *self.cacert* is replaced by another variable in the program.

```
Example1.1:
-  path = candidate
+  path = str(candidate)

Example1.2:
-  verify = self.cacert
+  verify = self.verify
```

### 4.3 (C2) Change of Method Call to a New Instance

Fix Pattern:

```
C2.1:Call New API
-  func.api1()
+  func.api2()

C2.2:Call New Function
-  func1
+  func2
```

**Description** Changing the buggy method invocation by modifying the method name or API name, which caused mainly by two reasons, one is a typo, and the other is that the original method signature is changed. When employing this pattern to fix bugs, we can apply other method names in the bug file to replace the buggy method call. If the correct instance is not in the bug file, the fixing fails.

```
Example:
-  os.link(dest, path)
+  os.symlink(dest, path)
```

### 4.4 (C3) Method Call with Different Number of Arguments

Fix Pattern:

```
C3.1:Add Arguments
-  func(arg1)
+  func(arg1,arg2,...)

C3.2:Delete Arguments
-  func(arg1,arg2,...)
+  func(arg1)
```

**Description** Adding or deleting argument(s) when the method signature has been changed or has the suitable overridden methods. When employing this add arguments pattern to fix

bugs, we traversed the bug file, listed all variables and instances as candidate arguments, and put the candidate arguments into a list. Generally, we added no more than three arguments by default. We selected elements from the candidate list in order and added them to the buggy code. If this generated patch is the same as the correct patch, it succeeds; otherwise, it fails. If all the potential arguments could not fix the bug successfully, the fix fails. For the delete arguments pattern, we deleted the current arguments in turn.

```
Example:
-  test_case.assertEqual(grad, d_input)
+  test_case.assertEqual(grad, d_input, 1e-4)
```

### 4.5 (C4) Method Call with Different Value of Arguments

Fix Pattern:

```
-  func(arg1)
+  func(arg2)
```

**Description** Replacing arguments in the method invocation with changing the value of arguments. The replacement object can be literals, variables, or other expressions. When employing this pattern, we can list all variables and instances in the bug file as candidate replacement objects. If the correct patch does not appear in the bug file, the fix fails.

```
Example:
-  os.unlink(self.tmplfile)
+  os.unlink(self.tmpfile)
```

### 4.6 (C5) Change Loop Variable to Iterable Object

Fix Pattern:

```
-  for key in var:
+  for key in list(var):
```

**Description** Changing the *for* statement loop variable to an iterable object. In Python syntax, *for* statement can directly traverse list, dict, and tuple objects without relying on index. For example, developer cast the return value of *result.keys()* to the list type.

```
Example:
-  for key in result.keys():
+  for key in list(result.keys()):
```

### 4.7 (C6) Change of Precondition Check

Fix Pattern:

```
C6.1:Change Retrieval Variable
-  if var1 in obj:
+  if var2 in obj:

C6.2:Change Constraint Condition
-  if condition1
+  if condition2
```

**Description** Changing the precondition of the if statement. This fix pattern can be divided into two sub-fix patterns: (1) Changing the retrieval variable in an expression, *obj* is usually a list, and *var* is a value to be retrieved; (2) Changing the whole constraint condition, *condition* can be a single variable used to determine whether it is null value. Most of the time, it is difficult for us to apply the change constraint condition pattern directly. Since the *condition* can also be a complex expression.

```
Example6.1:
-  if 'import_tasks' in task_ds:
+  if 'import_role' in task_ds:

Example6.2:
-  if found_line == line_number
+  if found_line == (line_number+1)
```

### 4.8 (C7) ADD/DEL Precondition Check

Fix Pattern:

```
C7.1:Add Precondition
-  if constraint1
+  if constraint1 and constraint2

C7.2:Delete Precondition
-  if constraint1 and constraint2
+  if constraint1

C7.3:Add Attribute Check
-  if obj
+  if obj and hasattr(obj,attr)
```

**Description** Adding or deleting the precondition(s) of the if statement. This fix pattern can be divided into three sub-fix patterns: (1) Adding at least one precondition; (2) Deleting precondition(s); (3) Adding the attribute check for an object. When employing the add preconditions pattern, we can traverse the bug file and list all variables and instances as candidates, adding new constraints via *and* keyword. For deleting precondition pattern, we can delete the preconditions in turn.

```
Example:
-  if view_func:
+  if view_func and hasattr(view_func, '__name__'):
```

### 4.9 (C8) Change of Exception Type

Fix Pattern:

```
C9.1:Change Exception Type
try:
    ...
-  except type as e:
+  except Exception as e:
    ...
```

**Description** Replacing the inappropriate except types in the *try/except* block. In this pattern, *type* is the Python's built-in Exception Type, while '...' means the subsequent statements dependent on the try/except. When employing this pattern, we applied common exception types (e.g., ValueError, IndexError, KeyError, TypeError, etc.) to replace the exception types in the bug code. However, due to many abnormal exception types and even user-defined types, it is difficult for us to repair accurately. Here, we used *Exception*, a general exception type, to fix it.

```
Example:
-  except shade.OpenStackCloudException as e:
+  except Exception as e:
```

### 4.10 (C9) Change of Third-party Library Dependency

Fix Pattern:

```
- lib1.func()
+ lib2.func()
```

**Description** Replacing the library name with another one that has the same API name. The bug fixing may be due to adding, deleting, or changing the import file. This pattern is suitable for batch repairs. We can replace the faulty library name by traversing the imported library name in the bug file. If the library name is a simple typo, then it can be easily fixed; if it is a code change due to other reasons such as system upgrade, then it is difficult to fix it.

```
Example:
- cgi.parse_qsl(qs, keep_blank_values=True):
+ parse_qsl(qs, keep_blank_values=True):
```

### 4.11 (C10) Change of Error Messages Content

Fix Pattern:

```
- raise ValueError('s1')
+ raise ValueError('s2')
```

**Description** Since this bug fixing mainly repairs typos of error messages, we can not infer the correct patch. Nevertheless, this pattern is suitable for batch repairs, where the correct text or correct formatting is known to replace the log text throughout the system.

```
Example:
- raise ValueError("'fun' must return at most 1-d array_like.")
+ raise ValueError("'fun' must return at most 1-d array_like. "
+             "f0.shape: 0".format(f0.shape))
```

### 4.12 (C11) Change of With Method Call

Fix Pattern:

```
C11.1:Change Method Call
- with exp1:
+ with exp2:

C11.2:Change read/write parameters
- with open(var, arg1) as f:
+ with open(var, arg2) as f:
```

**Description** Changing the *with* statement context expression. There is a special case in this pattern, that is, to fix the read/write parameters in *open()*.

```
Example:
- with open(local_file, 'r') as f:
+ with open(local_file, 'rb') as f:
```

## 5 Python Fix Pattern

### 5.1 Answer to RQ2: Python Specific Fix Pattern

We summarized 18 Python fix patterns by automatic clustering and manual summarization, which were classified into three categories: Python Operation (P1-P3), Dynamic Feature (P4-P10), and Python Migration (P11-P18) types, as shown in Table 5. We presented a catalog of fix patterns in Sections 5.2 and 5.3 and provided a description and example for each pattern. Example code with a prefix "-" is the bug code, while code with prefix "+" is the fixing code.

**Discussion** Python Operation pattern accounts for 46.7% (114/244), Dynamic Feature pattern accounts for 9.4% (23/224), and Python Migration pattern accounts for 43.% (107/244). Python operator patterns fix the misuse of equality operators or inequality operators; and dynamic feature patterns focus on repairing the misuse of the Python dynamic feature function. Also, we summarized migration patterns. Although the official manual has detailed Python migration rules, there are still bugs related to Python migration in practice. Therefore, we have summarized some of the Python migration fixes that are frequent occurrences in Appendix. The migration patterns are not used as often as other patterns, but they are an integral part of implementing automated repair tools. Unlike common fix patterns, Python

**Table 5** Python fix patterns

| No. | Pattern |
|---|---|
| (P1) | Compare Objects by (is) Operator |
| (P2) | Compare Strings by Equality Operator |
| (P3) | Change (in) to any() to Check Value in Generators |
| (P4) | Change (not all()) to any() to Check Value Present in Sequences |
| (P5) | Check Function Callable by callable() |
| (P6) | Delete None Check with hasattr() |
| (P7) | Add None Check with isinstance() |
| (P8) | Add Object Type Check with issubclass() |
| (P9) | Change type() to isinstance() to Check Object Type |
| (P10) | Change locals() Key Value Traversal |
| (P11) | Python 2.x Data Type Compatible with Python 3.x Data Type |
| (P12) | Python 2.x xrange() Compatible with Python 3.x range() |
| (P13) | Change Python 3.x map() Returned Value to List Type |
| (P14) | Python 3.x Check Dictionary has_key() |
| (P15) | Change Python 3.x Dictionary API Name |
| (P16) | Change Python 3.x Float Division to Integer Division |
| (P17) | Change Python 3.x super() Backward Compatibility |
| (P18) | Check Property Function by getattr() |

fix patterns only focus on Python's special operators and functions, which causes the use of Python fix patterns are far less than that of common fix patterns, but less use does not mean that they are not essential.

## 5.2 Python Operators (PO)

### 5.2.1 (P1) Compare Objects by (is) Operator

Fix Pattern:

```
- var == Object
+ var is Object
or
- obj
+ obj is not None
```

**Description** Replacing equality operator with *is* keyword. In Python, the equality operator compares two strings and checks whether the values are equal. Another representation of this fix pattern is to replace inequality operator with *'is not'*. For example, *None* is an object in Python, and we must use the *is* keyword to check whether the object is a null value.

```
Example:
- if (out_shapes == None):
+ if (out_shapes is None):
```

#### 5.2.2 (P2) Compare Strings by Equality Operator

Fix Pattern:

```
- 's' is str
+ 's' == str
```

**Description** Replacing *is* keyword with equality operator. In Python, *is* keyword is used to compare the memory addresses of two variables. Another representation of this fix pattern is to replace *'is not'* with inequality operator. For example, *'left'* is a string, we must use the equality operator to check whether the literal value of *direction* is *'left'*.

```
Example:
- if direction is 'left':
+ if direction == 'left':
```

#### 5.2.3 (P3) Change (in) to any() to Check Value in Generators

Fix Pattern:

```
- X in Y:
+ any(e is X for e in Y):
or
- X not in Y:
+ not any(e is X for e in Y):
```

**Description** Replacing *'X in Y'* with *'any(e is X for e in Y)'*. In Python, *in* operator is used to determine whether an element is in an iterable object (excluding generator). For example, we assume that *ranks_static* = (x for x in [1,2,None]). *'None not in ranks_static'* return False.

```
Example:
- if None not in ranks_static:
+ if not any(r is None for r in ranks_static):
```

### 5.3 Dynamic Features (DF)

#### 5.3.1 (P4) Change (not all()) to any() to Check Value Present in Sequences

Fix Pattern:

```
- not all()
+ any()
```

**Description** Replacing *any()* with *'not all()'*. *any()* returns True if any element of an iterable object is True, while *all()* returns True when all elements in the given iterable object are True. For example, the iterable object is *[True]*, *'not all(iterable)'* returns False and *any(iterable)* returns True. But, if the iterable object is *[True,False]*, the two expressions all return True. Another representation of this fix pattern is to replace *all()* with *'not any()'*.

```
Example:
- if not all(obj.pk for obj in objs):
+ if any(obj.pk is None for obj in objs):
```

#### 5.3.2 (P5) Check Function Callable by callable()

Fix Pattern:

```
- hasattr(obj, '__call__')
+ callable(obj)
```

**Description** Replacing *callable(obj)* with *hasattr(obj,'__call__')*. In practise, *hasattr()* return more false positives than *callable()*. For example, we assume that *pd_index* is an instance of the Test class.

```
class Test(object):
    def __getattr__(self,name):
        return  name

pd_index = Test()

if hasattr(pd_index,'__class__'):
    print('1')
if callable(pd_index):
    print('2')

Output:1
```

When use *dir()* to detect the attributes of *pd_index*. We could find that *pd_index* does not have *__call__()*.

```
Example:
-  if ((pd_index is not None) and hasattr(pd_index, '__call__')):
+  if ((pd_index is not None) and callable(pd_index)):
```

#### 5.3.3 (P6) Delete None Check with hasattr()

Fix Pattern:

```
-  (obj is not None) and hasattr()
+  hasattr()
```

**Description** Deleting None check when calling hasattr(). This is an unnecessary check. For example, when the *loader* object is *None*, *hasattr(loader, "get_source")* returns *False*. Developer remove *None* checks before *hasattr()*.

```
Example:
-  if loader is not None and hasattr(loader, "get_source"):
+  if hasattr(loader, 'get_source'):
```

#### 5.3.4 (P7) Add None Check with isinstance()

Fix Pattern:

```
-  not isinstance(X,type)
+  X is not None and not isinstance(X,type)
```

**Description** Adding None check when calling *isinstance(object, type)* In Python, *None* keyword is an object. Also, it is a data type of the class *NoneType*. For example, if *kvstore* is *None*, *'not isinstance(kvstore,type)'* returns True. This bug is a high-frequency error in Python programs.

```
Example:
- if not isinstance(kvstore, KVStore):
+ if kvstore is not None and not isinstance(kvstore, KVStore):
```

#### 5.3.5 (P8) Add Object Type Check with issubclass()

Fix Pattern:

```
- issubclass()
+ isinstance(Func, type) and issubclass()
```

**Description** Adding object type check when calling *'issubclass()'*. *'issubclass(class, classinfo)'* checks whether the *class* argument is a subclass (direct, indirect or virtual) of classinfo. In Python 3.7, the method requires the class argument to be a class type.

```
Example:
- if issubclass(Func, function.Macro):
+ if (isinstance(Func, type) and
+   issubclass(Func, function.Macro)):
```

#### 5.3.6 (P9) Change type() to isinstance() to Check Object Type

Fix Pattern:

```
- type(obj) == type
+ isinstance(obj, type)
```

**Description** Replacing *type()* with *instance()* when checking the object data type. *type(object)* returns the type of an object, and *isinstance(object, classinfo)* checks whether the *object* is an instance or subclass of *classinfo*. For example, developer uses *instance()* to detect whether *v* is a list, instead of calling *type()*.

```
Example:
- if type(v) == list:
+ if isinstance(v, list):
```

#### 5.3.7 (P10) Change locals() Key Value Traversal

Fix Pattern:

```
- locals().keys()
+ locals().copy()
```

**Description** Replacing *locals().keys()* with *locals().copy()* to traverse the local symbol table key value. The local symbol table is a data structure (dictionary) maintained by a compiler that contains all necessary information about the program. It is not an ordinary dictionary type data. If we want to traverse this dictionary, we can call locals().copy() to create a copy.

```
Example:
- locals_var = locals().keys()
- for name in locals_var:

+ locals_var = locals().copy()
+ for name, val in locals_var.items():
```

## 6 Evaluating Fix Patterns

### 6.1 Usage of Fix Patterns

In this section, we described the usage of fix pattern. In our data set, each bug report contains both the bug file and the correct file. By comparing the two versions of the files, we can obtain the bug-fixing code changes. We consider the bug code as the **fix target**, the fixing code as the **correct patch**, and the code generated by fix pattern as the **generated patch**. We attempted to apply the fix patterns in RQ1 and RQ2 to repair bugs and compared the generated patches with correct patches to evaluate the fix pattern effectiveness. We described pattern matching, patch generation, and patch verification as follows in detail.

**Pattern Matching** We transformed the bug code into AST and matched the appropriate pattern according to its AST type. For the following example (#ansible-15), the bug code AST type is *ast.Assign*, and the pattern that matches its type in our fix patterns happens to be (C1).

```
bug code: input_shape=(input_shape[0],input_shape[1],output_dim)
bug code AST: [<_ast.Assign object at 0x7fe5802628d0>]
Fix Pattern:  (C1) Change of Assignment Expression
```

**Patch Generation** As shown in Fig. 6, patch generation is divided into two types. The first fixing method does transform the bug code according to the semantics of the fixing code in fix pattern without changing the objects (variables, parameters, or functions), as shown in Fig. 6(a). This fixing process can be implemented at the string level, with high execution efficiency. The second fix method involves changing the buggy objects (variable, parameter, function). We traversed the bug file and list candidate objects: all variables (global variables, member variables), parameters, and methods (global methods, member methods). Then, we applied the candidate objects to replace the buggy objects in turn and transformed the bug

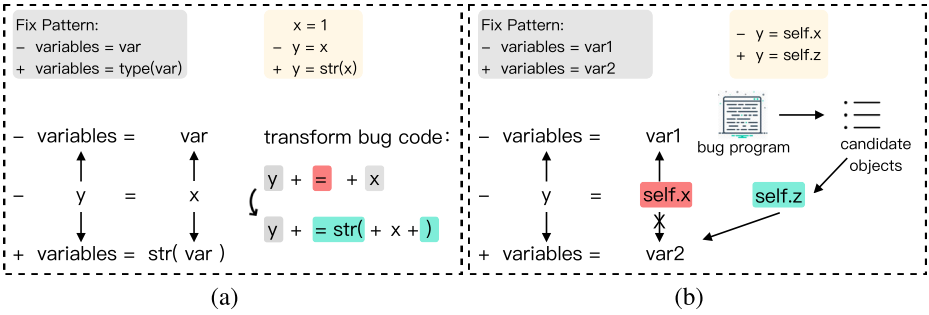



**Fig. 6** Patch generation of fix patterns

code according to the semantics of the fixing code in the pattern, as shown in Fig. 6(b). This fixing process performs at the AST level. After fixing, we parsed the fixing AST into the fixing code.

**Patch Validation** If the generated patch is identical to the correct patch, the fix is successful. Otherwise, according to Chakraborty et al. (2018), we selected the generated patch whose edit distance is less than 3 for manual review, to identify the generated patch which is not identical but semantically equivalent with the correct patch. Chakraborty et al. (2018) divided the tree edit distances into three sections: (small: edit size = 1, medium: $2 \leq$ edit size $\leq 5$, and large: $6 \leq$ edit size $\leq 10$) and smaller patches (1-3 edit sizes) is better than the larger ones. Chakraborty et al.'s study found that generated patches with a large edit distance larger than 3 tend to contain completely different semantic information. Thus, we chose to set the threshold into 3. We identify the generated patch as failed repair if it is not identical or semantically equivalent with the correct patch.

### 6.2 Answer to RQ3: Effectiveness of Fix Pattern for Fixing Single-hunk Bugs

We attempted to apply the fix pattern to fix single-hunk bugs in BugsInPy (contain 493 bugs from 17 real-world Python programs) (Widyasari et al. 2020) and QuixBugs (contain 40 buggy algorithmic programs) (Lin et al. 2017). These two benchmarks are used for the study of defect prediction (Akimova et al. 2021) and automatic program repair (Ye et al. 2021). We obtained 101 single-hunk bug reports from the two benchmarks, 61 bug reports in BugsInPy and 40 bug reports in QuixBugs.

**Discussion** Table 6 shows the evaluation results; the fix pattern can match 34 bugs; among them, 10 patches are identical, 3 patches are semantically equivalent, and 21 patches are plausible but incorrect.

In the 10 correctly fixed patches, 5 of them changed the arguments in API. The 3 semantically identical patches all fix exception types in try/except blocks, and the patch we generated is for the general type `Exception`. Among the 21 plausible patches, most of them are adding `if` precondition or modifying the right-side expression in the assignment statement. Although all these 21 bugs only have one line of bug code, they involve in complicated control flow; thus, it is difficult to repair them only by replacing objects.

A total of 8 fix patterns were applied in the repairing. Pattern with the largest number of fixed bugs is C4, which has completely fixed 5 bugs. The pattern with the second number of fixed bugs is C1 (2 bugs have been fixed in total). This pattern matched 9 bugs and is

**Table 6** BugsInPy and QuixBugs single-hunk bugs fixed by fix patterns

| Benchmark | C1 | C2 | C3 | C4 | C6 | C7 | C8 | C11 |
|---|---|---|---|---|---|---|---|---|
| keras-6 | ○ | | | | | | | |
| keras-38 | | | ● | | | | | |
| keras-4 | | | ○ | | | | | |
| luigi-19 | | | | | | ○ | | |
| luigi-4 | | | | | | ○ | | |
| luigi-13 | | ● | | | | | | |
| luigi-25 | ○ | | | | | | | |
| cookiecutter-1 | | | | | | | | ○ |
| cookiecutter-3 | | | ○ | | | | | |
| scrapy-11 | ○ | | | | | | | |
| scrapy-37 | | | | | ○ | | | |
| scrapy-13 | ○ | | | | | | | |
| ansible-15 | | | | | | ● | | |
| tornado-11 | | | | | ○ | | | |
| tornado-14 | | | | | ○ | | | |
| matplotlib-27 | ● | | | | | | | |
| matplotlib-7 | | | | | ○ | | | |
| matplotlib-26 | | | | ● | | | | |
| matplotlib-24 | | | | ● | | | | |
| spacy-5 | | | | ● | | | | |
| pandas-69 | | | | | | ○ | | |
| pandas-89 | ○ | | | | | | | |
| pandas-116 | ○ | | | | | | | |
| pandas-74 | | | | | | ○ | | |
| pandas-10 | | | | | | ○ | | |
| pandas-30 | | | | | | | ⊕ | |
| pandas-64 | ○ | | | | | | | |
| pandas-48 | | | | | | | ⊕ | |
| pandas-76 | | | | | | | ⊕ | |
| pandas-78 | ● | | | | | | | |
| hanoi | | | | ● | | | | |
| rpn_eval | | | | ● | | | | |
| topological_ordering | | | | | ○ | | | |

●: patch is identical; ⊕: patch is semantically equivalent; ○: patch is plausible but not correct.

the pattern with the highest number of matches. In the remaining patterns, C2, C3 and C7 patterns fixed one bug respectively. However, pattern C7 matched a total of 6 bugs, which is only less than C1. C8 pattern generated three patches with the same semantics, which is because the fixing code in pattern is a common exception type. C6 and C11 patterns matched 5 bugs but failed to fix them.

**Table 7** The number of bugs and matched bugs for each repository in test set

| Repository | Bugs | *Single-hunk bugs* | *Complex bugs* | Matched bugs |
|---|---|---|---|---|
| pallets/flask | 22 | 5 | 17 | 9 |
| mopidy/mopidy | 8 | 0 | 8 | 4 |
| powerline/powerline | 6 | 3 | 3 | 1 |
| tensorlayer/tensorlayer | 11 | 0 | 11 | 7 |

It is considered that this is a reasonable result since most of the bugs in the benchmarks are functional bugs that fail in specific test cases. The experimental results show that these fix patterns can fix about 12.8% (13/101) of the single-hunk bugs in real-world development. Although the bug fixed rate of 12.8% is not relatively large, it demonstrates that bugs can be fixed independently without human intervention, which means that the fix patterns we proposed can directly help developers to perform white-box testing and screen out some of the bugs in the early stages of a project, thus reducing maintenance costs later in the project.

### 6.3 Answer to RQ4: Effectiveness of Fix Pattern in the wild

To evaluate the potential of these fix patterns in the wild, we investigated the following sub-questions. RQ4-1:How many single-hunk bugs can be matched, and how many single-hunk bugs can be fixed? RQ4-2: How effective are these fix patterns for fixing complex bugs?

We followed the criteria in Section 3.2 to select four open-source projects from GitHub as our test set, as shown in Table 7. From the four projects, we obtained a total of 109 bug reports. We specified the fixing granularity, i.e., no more than 10 changed files in each commit, and typo bugs were eliminated. Finally, we obtained 47 bug reports for experimental evaluation. Among the 47 bugs, 8 are single-hunk bugs, and the rest are **complex bugs**. We obtained the bug-fixing code changes for each bug, where the number of lines of bug code is shown in the second column of Table 8. And then, for each bug, we transformed the bug code into an AST line by line and matched the fix pattern according to its AST type. After matching, we tried to fix the bug code with the fixing method described in Section 6.1.

**Discussion** As shown in Table 8, we presented the effectiveness of fix pattern in the wild, which including each bug was fixed by which fix pattern, fixed rate, and matched rate. In our test set, 25% of (2/8) bugs can be matched by our fix pattern, the generated patches were plausible but not correct. For complex bugs, our fix pattern could match nearly half of (19/39) bugs. Among the 21 bugs, 8 were matched our fix patterns but could not be fixed, and 6 were partially fixed by our fix patterns. The average fixed rate is nearly 15%, while the average matched rate is 37%, as much as twice of average fixed rate, which means half matched patches could not be fixed.

This experimental result reflects how the fix pattern-based APR techniques apply in the real world. Although none of these 47 bugs have been completely fixed, the fix pattern still has a competitive fixed rate and matched rate for complex bugs. Moreover, there are 3 bug fixing that are done by two patterns, which shows that assemble patterns could be used for repair complex bug. On the other hand, we also attempted to disassemble the complex bug to match with our fix patterns. With these two ideas, the existing fixing method could be improved in the future.

**Table 8** Bugs in the wild fixed by fix patterns

| No. | del | C1 | C2 | C3 | C4 | C5 | C7 | C8 | C10 | C11 | P1 | P9 | Fixed | | Matched | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | ○ | | | | | | | | | 0 | *0.0%* | 1 | *100.0%* |
| 2 | 3 | | | | | | | | | ⊕ | | | 1 | *33.3%* | 0 | *0.0%* |
| 3 | 18 | | ● | | | | | | | | | | 5 | *27.8%* | 0 | *0.0%* |
| 4 | 9 | ○ | | | | | | | ○ | | | | 0 | *0.0%* | 2 | *22.2%* |
| 5 | 4 | | | ● | | | | | | | | | 1 | *25.0%* | 0 | *0.0%* |
| 6 | 4 | ○ | | | | | | | | | | | 0 | *0.0%* | 1 | *25.0%* |
| 7 | 1 | | | | | | ○ | | | | | | 0 | *0.0%* | 1 | *100.0%* |
| 8 | 3 | | | | | | | | ● | | | | 1 | *33.3%* | 0 | *0.0%* |
| 9 | 6 | | | ● | | | | | | | | | 1 | *16.7%* | 0 | *0.0%* |
| 10 | 13 | | | ○ | | | | | | | | | 0 | *0.0%* | 8 | *61.5%* |
| 11 | 4 | | | | ● | | | | | | | | 2 | *50.0%* | 0 | *0.0%* |
| 12 | 2 | | | | ● | | | | | | ○ | | 1 | *50.0%* | 1 | *50.0%* |
| 13 | 38 | ○ | | | ○ | | | | | | | | 0 | *0.0%* | 36 | *94.7%* |
| 14 | 1 | ○ | | | | | | | | | | | 0 | *0.0%* | 1 | *100.0%* |
| 15 | 7 | | | ○, ● | | | | | | | | | 1 | *14.3%* | 4 | *57.1%* |
| 16 | 15 | ○ | | ● | ● | | | | | | | | 3 | *20.0%* | 2 | *13.3%* |
| 17 | 25 | ○ | | | | | | | | | | | 0 | *0.0%* | 1 | *4.0%* |
| 18 | 2 | ○ | ○ | | ○, ● | | | | | | | | 1 | *0.0%* | 5 | *100.0%* |
| 19 | 88 | | | ○, ● | ○ | | | | | ○ | | | 9 | *10.2%* | 20 | *22.7%* |
| 20 | 25 | | | | | ○ | | | | | | ● | 1 | *4.0%* | 2 | *8.0%* |
| 21 | 14 | ● | | | | | | ⊕ | | | | | 4 | *28.6%* | 3 | *21.4%* |
| *avg.* | | | | | | | | | | | | | | *14.9%* | | *37.1%* |

**del**: means to delete *x* lines of buggy code;

**fixed**: means that the buggy code of *x* line is fixed correctly;

**matched**: means that the fix pattern matches the *x* line of buggy code, but the generated patch is not correct;

●: patch is identical; ⊕: patch is semantically equivalent; ○: patch is plausible but not correct.

## 7 Threats to Validity

Threats to internal validity are: (1) fix patterns we proposed may not be sufficient. In our experiments, we filtered out as many false positives and duplicate bug reports as possible. In addition, we restricted the granularity of bug-fixing code to ensure that fix patterns can be extracted accurately since our algorithm is mainly focused on single-hunk bugs; (2) the clustering approach may cause over-fitting. Other popular clustering methods (e.g., k-means, hierarchical clustering, etc.) may find more bug fix patterns. However, our approach finds highly similar bug-fixing codes by calculating the edit distance of the AST, and this approach preserves the code structure and can extract many of the most pervasive bug fix patterns; (3) we only focused on bug fixing for committed bugs. Bugs that are fixed before a commit is merged into the master branch are not found by our method.

Threats to external validity are: (1) false positives in bug data set. We identified bugs by looking for keywords that are synonymous with bugs, such as wrong, defect, fault, etc., however, the description of bug report may not be directly related with the bug, sometimes

we could not guarantee the link between commits and the bug reports. False positives may be introduced without a strict and formal definition of the bug constitution; (2) When we applied the proposed fix pattern to other Python projects which are not included in our research, some patterns may not be capable. This is because some Python patterns may be specific to our subject systems.

## 8 Related Work

### 8.1 Automated Program Repair

Automated Program Repair (APR) is an emerging technology that could effectively reduce software maintenance costs, a popular academic research topic in recent years.

**Search-based APR** collects the existing modifications of the program as a potential patch space. The heuristic methods or evolutionary algorithms are employed to find the suitable patch from the potential patch space. Generating program patches is regarded as finding the optimal solution from the search space.

GenProg (Weimer et al. 2009) is an iconic search-based APR method, which searches for the correct patch by a genetic algorithm. GenProg extracts programs as abstract syntax trees and code segments as sub-trees. Genetic operation is to select two programs with high fitness and exchange the mutation operations among them. In 2012, Le Goues et al. (2012) conducted a large-scale experiment, and they selected 105 defects to repair and successfully repaired 55 of the 105 defects. Weimer et al. (2013) improved the GenProg method and proposed the AE. To reduce the number of generated candidate patches, AE proposes the concept of equivalence classes. By setting a series of rules, we can quickly check specific types of equivalent transformations and avoid them during the modification process. The validation results show that the repair time cost was one-third of the original one. Qi et al. (2015) conducted manual inspections on the repairs of 105 defects in GenProg and AE. There were only two in Genprog, and only three in AE repair results were semantically correct. The research results show that the existing repair technique had a low repair rate and cannot guarantee the correctness of the repair results. Hua et al. (2018) proposed SketchFix, which is a solution to the low search efficiency problem of search-based APR, by generating candidate fixes on demand (as needed) during test execution. SketchFix effectively fix bugs in expression manipulation at the AST node-level granularity compared to other APR techniques on the Defects4J benchmark. Within the default settings, SketchFix fixed 19/357 bugs in 23 minutes on average. Wen et al. (2018) proposed CapGen, an automatic generation technique for context-aware patches. CapGen uses the more fine-grained AST context information to generate patches, proposes three models to get more fine-grained patch repair materials, and uses context-aware information to sort the mutation operations, limiting the search space. The experimental results show that CapGen can reach 84% accuracy while filtering out 98.78% of suspected correct patches.

Search-based APR is simple and intuitive; however, we may not find the correct patch within a limited time threshold. One reason is that the candidate patch search space may not contain the correct patch itself; another reason is that the search space is too large, and the search space explosion problem greatly compromises the APR tools' execution efficiency (Long and Rinard 2016).

**Semantic-based APR** uses semantic information to synthesize repair patches through symbolic execution and constraint solving.

Nguyen et al. (2013b) proposed the C language constraint-based repair algorithm, which uses a combination-based program synthesis method to generate patches. Mechtaev et al. (2015) proposed DirectFix, an automatic repair method to generate simplified patches for the core problem of how to generate high-precision patches. DirectFix no longer considers enumerating candidate patches for each suspected bug location but merges bug location and patch generation more efficiently, solves it as part of the maximum satisfiable problem, and directly selects the most simplified patch that meets the constraints as the output. DirectFix experiments on the software-artifact infrastructure repository data set (Do et al. 2005) and the Coreutils data set (Cadar et al. 2008). DirectFix can generate 59% of bug patches, of which 56% are correct patches; the correct patches output by DirectFix are extensive and are simpler than SemFix. Mechtaev et al. (2016) proposed Angelix, a lightweight symbolic execution technique to deal with bugs in more extensive programs. Compared with the previous similar repair techniques SemFix and DirectFix, the symbolic execution of this method is more lightweight. Angelix can perform on multiple buggy locations, and it can automatically repair the famous heart bleed vulnerability. The experiment on GenProg Benchmark shows that the repair accuracy rate was 35.7%. Xuan et al. (2016) proposed a semantic-based repair method Nopol for the repair of incomplete specifications, focusing on conditional errors. Nopol is a particular method to repair the defects of if conditions and repairs the two common defects of conditional errors or missing conditional statements. Unlike the SemFix, Nopol only targets if conditional expressions with a small search space and can apply to large-scale programs.

Semantic-based APR is more efficient than search-based APR because the search space is more manageable by using program synthesis with restricted components. However, the effectiveness may be limited by the constraint solving and program synthesis capabilities. Moreover, both two ARP techniques suffer from overfitting problems (Xin and Reiss 2017).

### 8.2 Fix Pattern Mining

Many APR techniques exploit human patches (Kim et al. 2013; Long et al. 2017) or mined fix patterns (Liu et al. 2018; Liu et al. 2019a).

Pan et al. (2009) defined 27 Java bug fix patterns. These patterns were extracted based on the syntactic components and context of the source code involved in fixing the bug. After that, they manually analyzed the bug fixes in the open-source Java project and developed an extraction tool that can automatically identify bug fix patterns. This groundbreaking research laid the foundation for follow-up research. Kim et al. (2013) performed APR tool with common predefined fix patterns that contain only six patterns and can fix a few bugs. Later, Long et al. (2017) proposed Genesis, which can handle human patches and automatically infer code transformations in order to generate patches automatically. Genesis demonstrates the effectiveness of the inference algorithm, as well as the complete Genesis patch generation system. Liu et al. (2018) contributed the latest research results, where they proposed a method that uses convolutional neural networks to learn features and regroup similar instances by clustering. They evaluated the usefulness of the identified fix patterns by applying them to unfixed violations. Also, Liu et al. (2019a) built AVATAR, an APR system that exploits static analysis of the fix patterns of violations as an ingredient to generate patches. Their study highlights the relevance of static APR tools as indirect contributors to fixing ingredients that address code defects identified with functional test cases.

Besides the fix pattern research based on Java language, there are various researches on Linux system (Hong and Kim 2013), Cloud Computing Platform (Cotroneo et al. 2019), and

JavaScript language (Hanam et al. 2016). Hong and Kim (2013) derived findings on Linux concurrency errors from a review of changelog files for Linux version 2.6.x. They developed a pattern-based concurrency bug detection framework that defines and matches various bug patterns. Based on previous bug reports, they defined four concurrency bug patterns with different synchronization mechanisms that effectively detect new bugs in lock-based analysis techniques in Linux. Cotroneo et al. (2019) proposed a method to characterize the changes in bug-fixing code changes, which involved not only the content of the changes in bug fixes but also the location of the changes. Their work is a detailed empirical study of the repetitive patterns of vulnerability remediation changes for three OpenStack projects. Hanam et al. (2016) practiced the comprehensive study of the prevalent vulnerability patterns in server-side JavaScript code. It is a novel technique for automatically learning the types of bug-fixing changes based on language structure changes.

As mentioned above, in the existing research, researchers have proposed fix patterns based on various application scenarios and multiple programming languages; however, there are still many barriers to the existing fix patterns in practice. First of all, most APR tools based on fix patterns target the single-hunk bug. This issue is caused by the existing fix pattern extraction method. Researchers use code similarity or code snippet clustering to discover common fixing patterns from historical fixes. This research approach is prone to interference from noisy data, i.e., in a real development environment, there may be many bug-unrelated codes in the context of bug statements. Second, the total number of fix patterns proposed in existing studies is small. This status limits the fixing ability of APR tools directly. The most widely studied patterns are java fix patterns, but Liu et al. (2019b) showed that there are only 15 types of java fix patterns (37 patterns in total), and the number of fix patterns of other languages is even smaller. As each programming language has its own syntax rules, it cannot interoperate directly at the string level. Third, the existing research is focused on a few open-source benchmarks and does not take full advantage of the resources of the open-source community. In the future, with the construction and improvement of the open-source community, big code could promote the development of APR tools based on fix patterns.

## 9 Conclusion

In this study, we investigated bug reports collected from the open-source platform GitHub. We researched the recurring bugs and their fixing codes to extract the fix patterns for the corresponding bug types. The fix patterns contributed in this paper can help developers improve coding quality, provide insights about bug fixing, and support researchers in developing automation tools. Meantime, we proposed a novel technique for automatically mining fix patterns based on fine-grained bug-fixing code changes. We identified 29 fix patterns, of which there are 11 common fix patterns and 18 Python-special patterns in total. We applied the patterns to detect single-hunk bugs in two benchmarks and fixed 13 (13/101) bugs. Also, we evaluated the effectiveness of fix patterns in the wild. Our results provide meaningful contributions to improve the state-of-the-art automatic bug detection technical. As further work, we plan to combine fix pattern mining with automated program repair techniques to generate bug fixes automatically. In actual research, we also found some exciting findings like some fix patterns-special to the project, bug type change with the version evolution. These findings could prompt us further to explore the application and value of automated program repair.

# Appendix A

## A.1 (P11) Python 2.x Data Type Compatible with Python 3.x Data Type

Fix Pattern:

```
- type
+ six.type
```

**Description** This fix pattern is applied to fix the issue of data type incompatibility. Replacing *type* with *six.type*. Six library is the Python 2.x to 3.x compatibility library, which provides utility functions for smoothing over the differences between the Python versions.

```
Example:
- assert isinstance(org_id, int)
+ assert isinstance(org_id, six.integer_types)
```

## A.2 (P12) Python 2.x xrange() Compatible with Python 3.x range()

Fix Pattern:

```
- xrange()
+ range()
```

**Description** This fix pattern is applied to fix the issue of *range()* incompatibility. Replacing *xrange()* with *range()*. Python 3.x no longer supports *xrange()*.

```
Example:
- for epoch in xrange(self.n_epochs):
+ for epoch in range(self.n_epochs):
```

## A.3 (P13) Change Python 3.x map() Returned Value to List Type

Fix Pattern:

```
- map()
+ list(map())
```

**Description** This fix pattern converts the return value of *map()* into a list type. For example, *codelens* is a list type in Python 2.x, but it is a *map* object in Python 3.x. *numpy* which can not calculate the *map* object.

```
Example:
-  codelens = map((lambda x: len(x)), all_codes)
+  codelens = list(map((lambda x: len(x)), all_codes))
   last = numpy.product(codelens) - 1
```

### A.4 (P14) Python 3.x Check Dictionary has_key()

Fix Pattern:

```
- dict.has_key()
+ key in dict
```

**Description** Replacing *has_key()* with *'key in dict'*. Python 3.x deletes *has_key()* and replaces it with *'in'*, which is $Key\ in\ dict$.

```
Example:
- if (not self._parsed_data.has_key(key)):
+ if (key not in self._parsed_data):
```

### A.5 (P15) Change Python 3.x Dictionary API Name

Fix Pattern:

```
- dict.iterkeys()
+ dict.keys()
```

**Description** Replacing *iterkeys()* with *keys()* when calling dictionary. In Python 3.x, both $iter*$ and $view*$ methods correspond to $keys()$, $values()$.

```
Example:
- for key in kwargs.iterkeys():
+ for key in kwargs.keys():
```

### A.6 (P16) Change Python 3.x Float Division to Integer Division

Fix Pattern:

```
- x / y
+ x // y
```

**Description** This fix pattern changes division operation. Replacing *'/'* with *'//'*. For example, when *n* is an integer, n/4 is *int* type in Python 2.x while float type in Python 3.x.

```
Example:
- slopes = np.random.normal(size=((n / 4), 2))
+ slopes = np.random.normal(size=((n // 4), 2))
```

### A.7 (P17) Change Python 3.x super() Backward Compatibility

Fix Pattern:

```
- super().func()
+ super(SubClass,self).func()
```

**Description** This fix pattern changes the *super()* syntax. In Python 3.x, *super()* allows users to explicitly refer to the parent class. However, if we want to call *super()* in Python 2.x, we must use *super(SubClass,self)*.

```
Example: def on_group(self, *largs):
- super().on_group(*largs)
+ super(CheckBox, self).on_group(*largs)
```

### A.8 (P18) Check Property Function by getattr()

Fix Pattern:

```
- hasattr(obj,name)
+ getattr(obj,name[,default])
```

**Description** This fix pattern changes *'hasattr(object, name)'* to *'getattr(object, name [,default])'*. In early versions of Python 2.x, *hasattr()* had a bug about property function, as follows:

```
class Foo(object):
    @property
    def bar(self):
        raise SyntaxError

    def baz(self):
        raise SyntaxError

foo = Foo()

Output:
sys.version '2.7.x'
hasattr(foo, 'bar') return False
hasattr(foo, 'baz') return True`
```

Although this bug has been fixed in later versions. When writing mixed code compatible with Python 2.x and 3.x, users should pay more attention to this bug.

```
Example:
- if not hasattr(module, '__file__'):
+ if getattr(module, '__file__', None) is None:
```

**Acknowledgements** We thank the anonymous reviewers for their constructive comments. This work is partially supported by the the National Natural Science Foundation of China (No.62172209), the Key Program of the National Natural Science Foundation of China (No.61832009) and Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (No. YBN2019105178SW23).

## References


Åkerblom B, Stendahl J, Tumlin M, Wrigstad T (2014) Tracing dynamic features in python programs. In: Proceedings of the 11th working conference on mining software repositories. pp 292–295

Akimova EN, Bersenev AY, Deikov AA, Kobylkin KS, Konygin AV, Mezentsev IP, Misilov VE (2021) A survey on software defect prediction using deep learning. Mathematics 9(11):1180

Cadar C, Dunbar D, Engler DR et al (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol 8, pp 209–224

Chakraborty S, Allamanis M, Ray B (2018) Tree2tree neural translation model for learning source code changes. arXiv:181000314

Chen Z, Ma W, Lin W, Chen L, Li Y, Xu B (2018) A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of python dynamic features. Science Chin Inf Sci 61(1):012107

Cotroneo D, De Simone L, Iannillo AK, Natella R, Rosiello S, Bidokhti N (2019) Analyzing the context of bug-fixing changes in the openstack cloud computing platform, IEEE

Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10(4):405–435

Durieux T, Cornu B, Seinturier L, Monperrus M (2017) Dynamic patch generation for null pointer exceptions using metaprogramming. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 349–358

Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychol Bull 76(5):378

Fluri B, Wursch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans Softw Eng 33(11):725–743

Fluri B, Giger E, Gall HC (2008) Discovering patterns of change types. In: 2008 23rd IEEE/ACM international conference on automated software engineering. IEEE, pp 463–466

Habib A, Pradel M (2018) How many of all bugs do we find? a study of static bug detectors, IEEE

Hallgren KA (2012) Computing inter-rater reliability for observational data: an overview and tutorial. Tutor Quant Methods Psychol 8(1):23

Hanam Q, Brito FSdM, Mesbah A (2016) Discovering bug patterns in javascript. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. pp 144–156

Higo Y, Hayashi S, Hata H, Nagappan M (2020) Ammonia: an approach for deriving project-specific bug patterns. Empir Softw Eng :1–29

Hindle A, Barr ET, Gabel M, Su Z, Devanbu P (2016) On the naturalness of software. Commun. ACM 59(5):122–131

Holkner A, Harland J (2009) Evaluating the dynamic behaviour of python applications. In: Proceedings of the thirty-second australasian conference on computer science-volume 91, pp 19–28

Hong S, Kim M (2013) Effective pattern-driven concurrency bug detection for operating systems. J. Syst. Softw. 86(2):377–388

Hu M, Zhang Y (2020) The python/c api: Evolution, usage statistics, and bug patterns, IEEE

Hua J, Zhang M, Wang K, Khurshid S (2018) Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th international conference on software engineering. pp 12–23

Jiang L, Su Z (2009) Automatic mining of functionally equivalent code fragments via random testing. In: Proceedings of the eighteenth international symposium on Software testing and analysis. pp 81–92

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches, IEEE

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each, IEEE

Lin D, Koppel J, Chen A, Solar-Lezama A (2017) Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity, pp 55–56

Liu K, Kim D, Bissyandé TF, Yoo S, Le Traon Y (2018) Mining fix patterns for findbugs violations. IEEE Trans Softw Eng

Liu K, Koyuncu A, Kim D, Bissyandé TF (2019a) Avatar: Fixing semantic bugs with fix patterns of static analysis violations, IEEE

Liu K, Koyuncu A, Kim D, Bissyandé TF (2019b) Tbar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis. pp 31–42

Liu X, Zhong H (2018) Mining stackoverflow for program repair, IEEE

Long F, Rinard M (2016) An analysis of the search spaces for generate and validate patch generation systems, IEEE

Long F, Amidon P, Rinard M (2017) Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. pp 727–739
Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: Looking for simple program repairs. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1. IEEE, pp 448–458
Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th international conference on software engineering. pp 691–701
Monat R, Ouadjaout A, Miné A (2020), Static type analysis by abstract interpretation of python programs. ECOOP (LIPIcs). To appear
Negara S, Codoban M, Dig D, Johnson RE (2014) Mining fine-grained code changes to detect unknown change patterns. In: Proceedings of the 36th international conference on software engineering. pp 803–813
Nguyen HA, Nguyen AT, Nguyen TT, Nguyen TN, Rajan H (2013a) A study of repetitiveness of code changes in software evolution, IEEE
Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013b) Semfix: Program repair via semantic analysis, IEEE
Noda K, Nemoto Y, Hotta K, Tanida H, Kikuchi S (2020) Experience report: How effective is automated program repair for industrial software?, IEEE
Nugroho YS, Hata H, Matsumoto K (2020) How different are different diff algorithms in git? Empir. Softw. Eng. 25(1):790–823
Pan K, Kim S, Whitehead EJ (2009) Toward an understanding of bug fix patterns. Empir. Softw. Eng. 14(3):286–315
Pawlik M, Augsten N (2015) Efficient computation of the tree edit distance. ACM Trans Database Syst (TODS) 40(1):1–40
Pawlik M, Augsten N (2016) Tree edit distance: Robust and memory-efficient. Inf. Syst. 56:157–173
Qi Z, Long F, Achour S, Rinard M (2015) 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. pp 24–36
Saha S et al (2019) Harnessing evolution for multi-hunk program repair. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 13–24
Sanner MF et al (1999) Python: a programming language for software integration and development. J Mol Graph Model 17(1):57–61
Seaman CB (1999) Qualitative methods in empirical studies of software engineering. IEEE Trans Softw Eng 25(4):557–572
Shrout PE, Fleiss JL (1979) Intraclass correlations: uses in assessing rater reliability. Psychol Bull 86(2):420
Tratt L (2009) Dynamically typed languages. Adv. Comput. 77:149–184
Van Rossum G, Drake FL Jr (1995) Python tutorial, vol 620. Centrum voor Wiskunde en Informatica Amsterdam
Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. pp 805–816
Wang B, Chen L, Ma W, Chen Z, Xu B (2015) An empirical study on the impact of python dynamic features on change-proneness. In: SEKE. pp 134–139
Wang Y, Meng N, Zhong H (2018) An empirical study of multi-entity changes in real bug fixes, IEEE
Weimer W, Nguyen T, Le GouesC, Forrest S (2009) Automatically finding patches using genetic programming. In: 2009 IEEE 31st international conference on software engineering. IEEE, pp 364–374
Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: Models and first results, IEEE
Wen M, Wu R, Cheung SC (2016) Locus: Locating bugs from software changes, IEEE
Wen M, Chen J, Wu R, Hao D, Cheung SC (2018) Context-aware patch generation for better automated program repair, IEEE
Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y, et al. (2020) 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. pp 1556–1560
Xia X, Wan Z, Kochhar PS, Lo D (2019) How practitioners perceive coding proficiency, IEEE
Xin Q, Reiss SP (2017) Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis. pp 226–236
Xu Z, Liu P, Zhang X, Xu B (2016) Python predictive analysis for bug detection. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. pp 121–132

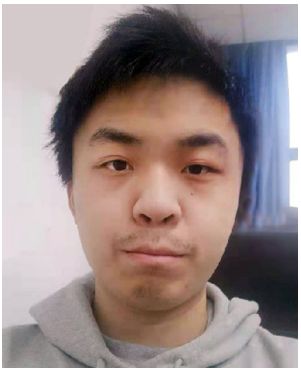
Xuan J, Martinez M, Demarco F, Clement M, Marcote SL, Durieux T, Le Berre D, Monperrus M (2016) Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. 43(1):34–55

Ye H, Martinez M, Durieux T, Monperrus M (2021) A comprehensive study of automatic program repair on the quixbugs benchmark. J. Syst. Softw. 171:110825

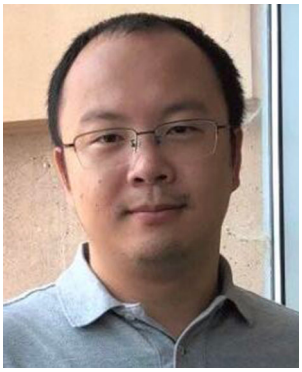
Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. pp 129–140



**Yilin Yang**



**Tianxing He**



**Yang Feng**

**Shaoying Liu**

**Baowen Xu**

## Affiliations

**Yilin Yang[1] · Tianxing He[1] · Yang Feng[1] · Shaoying Liu[2] · Baowen Xu[1]**

Yilin Yang
yilin.yang@smail.nju.edu.cn

Tianxing He
mf20320060@smail.nju.edu.cn

Shaoying Liu
sliu@hiroshima-u.ac.jp

Baowen Xu
bwxu@nju.edu.cn

[1] State Key Laboratory for Novel Software Technology, Nanjing University, No. 22 Hankou Rd., Gulou District, Nanjing, Jiangsu 210093, People's Republic of China

[2] Graduate School of Advanced Science and Engineering, Hiroshima University, Higashihiroshima, Japan