SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2018, Vol. 61 012107:1–012107:18
https://doi.org/10.1007/s11432-017-9153-3

A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features

Zhifei CHEN, Wanwangying MA, Wei LIN, Lin CHEN*, Yanhui LI & Baowen XU*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093*, China*

Received 18 February 2017/Accepted 19 June 2017/Published online 12 December 2017

Abstract Dynamic features in programming languages support the modification of the execution status at runtime, which is often considered helpful in rapid development and prototyping. However, it was also reported that some dynamic feature code tends to be change-prone or error-prone. We present the first study that analyzes the changes of dynamic feature code and the roles of dynamic features in bug-fix activities for the Python language. We used an AST-based differencing tool to capture fine-grained source code changes from 17926 bug-fix commits in 17 Python projects. Using this data, we conducted an empirical study on the changes of dynamic feature code when fixing bugs in Python. First, we investigated the characteristics of dynamic feature code changes, by comparing the changes between dynamic feature code and non-dynamic feature code when fixing bugs, and comparing dynamic feature changes between bug-fix and non-bugfix activities. Second, we explored 226 bug-fix commits to investigate the motivation and behaviors of dynamic feature changes when fixing bugs. The study results reveal that (1) the changes of dynamic feature code are significantly related to bug-fix activities rather than non-bugfix activities; (2) compared with non-dynamic feature code, dynamic feature code is inserted or updated more frequently when fixing bugs; (3) developers often insert dynamic feature code as type checks or attribute checks to fix type errors and attribute errors; (4) the misuse of dynamic features introduces bugs in dynamic feature code, and the bugs are often fixed by adding a check or adding an exception handling. As a benefit of this paper, we gain insights into the manner in which developers and researchers handle the changes of dynamic feature code when fixing bugs.

Keywords Python, fine-grained code changes, change behaviors, dynamic features, bug fixing

Citation Chen Z F, Ma W W Y, Lin W, et al. A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features. Sci China Inf Sci, 2018, 61(1): 012107, https://doi.org/10.1007/s11432-017-9153-3

1 Introduction

Dynamic features in programming languages support the modification of the execution status at runtime, for example, by dynamically deleting attributes of classes or instances, or providing developers with reflective interfaces. The succinctness, expressivity, and flexibility of dynamic features make them appealing for rapid development. The usages of dynamic features have been studied for several languages, including Python [1, 2], Java [3], JavaScript [4, 5], Smalltalk [6], and AspectJ [7]. Although these dynamic features are mentioned by some developers as useful constructs, they also pose threats to program comprehension and software maintenance. Wang et al. [8] reported that Python dynamic features are error-prone and

* Corresponding author (email: lchen@nju.edu.cn, bwxu@nju.edu.cn)

generally unpleasant to maintain, which makes dynamic feature code changeable in history. Park et al. [9] pointed out that extremely dynamic and functional features of JavaScript make it difficult to analyze web applications statically, particularly resulting in too many false positives. Consequently, we infer that dynamic features exhibit both benefits and costs in software development and maintenance.

This paper focuses on dynamic features in the Python programming language. Python, designed by Guido van Rossum in 1990 [10], is a popular dynamic programming language that contains a vast majority of dynamic features. Currently, Python is used everywhere from web services to scientific computation. In 2016, it won the top third programming language [1)]. Real-world Python programs commonly leverage built-in dynamic features in the entire execution phase, but the dynamic behavior is difficult to type [1]. Additionally, dynamic feature code in Python programs tends to be changeable in the history of maintenance activities [8]. However, few studies clearly demonstrate the manner in which dynamic features behave in software maintenance. Intuitively, dynamic features can be introduced to help fix bugs on the one hand, but on the other hand, the misuse of dynamic features may produce bugs. To this end, this paper characterizes the changes of Python dynamic feature code when fixing bugs to analyze the benefits and costs of dynamic features in fixing bugs.

To capture dynamic feature code changes in Python projects, an AST (abstract syntax tree) differencing tool was used to obtain fine-grained source code changes from 17926 bug-fix commits in 17 projects. First, we investigated the characteristics of dynamic feature code changes in practice, by comparing the changes between dynamic feature code and non-dynamic feature code when fixing bugs, and comparing dynamic feature changes between bug-fix activities and non-bugfix activities. Furthermore, we explored 226 bug-fix commits to investigate the motivation and the behaviors of dynamic feature changes when fixing bugs. The findings from this study include: (1) isinstance, getattr, hasattr, and super experience more changes than other dynamic features during bug-fix activities; (2) among the three change types, dynamic feature code is inserted or deleted more frequently compared with non-dynamic feature code when fixing bugs; (3) dynamic feature code is changed more frequently during bug-fix activities compared with non-bugfix activities; (4) when fixing bugs, dynamic features are often introduced to avoid harmful operations by checking the object state or modifying the execution status; and (5) many bugs in dynamic feature code can be fixed by adding a check or adding an exception handling. The study extends our preliminary paper [11] by concentrating on bug-fix feature changes and exploring bug patterns and fixing patterns participating in dynamic features. We list several benefits of this study as follows.

The study provides insights on each analyzed dynamic feature. The benefits and costs of each feature we studied can be evaluated by change behaviors during bug-fix activities. Developers can refer to the study results to deal with the manner of changing dynamic feature code when fixing bugs. The results also provide new factors of adopting dynamic features in Python programs, considering the strong functionality and potential threats in the development and maintenance.

The study provides opportunities to detect the bugs caused by the misuse of dynamic features. The study investigates the relationships between different feature changes with bug-fix activities and discovers several error-prone dynamic features. After learning the bug patterns in dynamic feature code, detection tools can be developed to detect the bugs hidden in those error-prone dynamic features. Detecting bugs in dynamic feature code is essential as these bugs are always triggered only at runtime.

The study provides guidelines for automatic program repair. Many feature change behaviors in our corpus follow particular patterns. Change patterns of dynamic features when fixing bugs supply repair modes for automatic program repair. After identifying bug patterns in the erroneous code, the fixing solutions are recommended according to the change patterns of similar erroneous code in history.

The following section introduces the background of feature changes when fixing bugs. Section 3 introduces the setup of this study and the methods of analyzing dynamic feature code changes. Sections 4 and 5 report the results of this study. Then, Section 6 discusses the implications, application, and limitations of our study. Finally, Section 7 surveys related work and Section 8 gives our conclusion.

1) http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

Table 1 The categories of Python dynamic features

Introspection		Object changes	Code generation	Library loading
type	vars	setattr	compile	__import__
getattr	callable	delattr	eval	reload
hasattr	locals	del	exec	
isinstance	globals		execfile	
issubclass	super			

2 Changing dynamic feature code for fixing bugs

First of all, this section describes the motivation of this paper by introducing the background of dynamic features and an example of feature changes when fixing bugs.

2.1 The background of Python dynamic features

Dynamic languages perform many common programming operations at runtime, while static languages deal with them during compilation. These operations include modifying the type system, extending objects and definitions, or modifying and extending the program by adding additional code. It needs varying degrees of performance, complexity, and difficulty costs for static languages to emulate similar behaviors, but dynamic languages support direct constructs to make use of them. Following the existing researches on Python dynamic features [1,2,8], Table 1 shows four categories of dynamic feature constructs in Python programs.

Introspection. This is a category of constructs that examine the state of Python objects and the runtime environment by behavioral reflection. For example, developers can call the hasattr function to check the existence of an attribute according to its dynamically-determined name. This is especially convenient when the attribute is created under certain constraints or the attribute name varies following different execution paths. In fact, introspection allows developers to examine code entities to find out what they are, what they do, and what they know; e.g., using globals statement to access or modify global identifiers.

Object changes. The object system can be modified on demand at runtime in Python programs. setattr and delattr are used to append and delete a dynamically determined attribute, respectively. del is used to delete any variable and makes it unavailable afterwards. This means the structure of a Python class or instance is mutable and the set of variables in the domain is never certain. It is advisable to apply the introspection mechanism to handle such uncertainty.

Code generation. These constructs allow programmers to execute (eval, exec, execfile) or compile (compile) code by taking a string containing Python code. It returns the resulting value of an expression for eval, execution result of the statements for exec, and execution result of a Python file for execfile. Developers use compile to compile the source into a code or an AST object that can be executed by eval or exec.

Library loading. This category of constructs can import or reload arbitrary libraries dynamically. If the name of the library to be imported is determined by underlying hardware or user inputs, __import__ is often applied to defer decisions. For example, when the names of the library to be imported are different on Windows and Linux systems, __import__ accepts compatible library name string as the parameter after checking the operating system it is running on. Additionally, reload is used to load the library for the second time especially when the code of the library is changed during that period. These constructs make it feasible to import demanded libraries to the current module by specifying module name at any time.

2.2 Motivating example in bug fixing

Figure 1 shows a motivating example about changing dynamic feature code to fix bugs. Code Version 1 gives a code snippet that defines and calls the computeScore function. computeScore computes a final

Code Version 1	**Code Version 2**	**Code Version 3**

def computeScore(task1_score, task2_score):
  finalScore = task1_score+task2_score
  return finalScore
 	
print computeScore('40', '45')
 	
Type bug in calling computeScore('40', '45')
	def computeScore(task1_score, task2_score):
	  if **type(task1_score) == int and **
	  type(task2_score) == int:
	  finalScore = task1_score+task2_score
	  else:
	  finalScore=int(task1_score)+int(task2_score)
	  return finalScore
	
	print computeScore('40', '45')
	print computeScore(40.5, 45.5)
	
	Type bug fixed in calling computeScore('40', '45'), but precision loss in calling computeScore(40.5, 45.5)
		def computeScore(task1_score, task2_score):
		  if **type(task1_score) in [int,float] and **
		  type(task2_score) in [int,float]:
		  finalScore = task1_score+task2_score
		  else:
		  finalScore=int(task1_score)+int(task2_score)
		  return finalScore
		
		print computeScore('40', '45')
		print computeScore(40.5, 45.5)
		
		Bugs fixed in both function calls

Figure 1 Motivating example.

score by adding the scores of two tasks, task1_score and task2_score. Owing to the lack of type declarations in Python, developers cannot limit the types of task1_score and task2_score. When executing computeScore ($'40',' 45'$), a type bug occurs at the second line because task1_score and task2_score accept string values instead of numeric scores. Generally, limited control of object state comes with the high flexibility of Python. This becomes especially challenging in the development of large-scale software. Code Version 2 fixes this type bug by introducing type to check the types of task1_score and task2_score, thus finalScore is correctly computed by explicitly converting the string into an int at the sixth line when calling computeScore ($'40',' 45'$). However, another bug appears after a new developer inserts computeScore $(40.5, 45.5)$ at the last line. task1_score and task2_score accept float scores but they are converted into int at the sixth line, thus a wrong finalScore 85 is returned. This reveals that although the introduction of type aids fixing type bugs, potential danger may also lie in type code. To fix unexpected floating point precision loss, Code Version 3 updates type code to accept both int and float at the second line, and thus finalScore 86.0 is correctly returned.

A first glance at the two changes of type for fixing bugs reveals both benefits and costs of dynamic features in fixing bugs. This example motivates our study of analyzing change behaviors of dynamic features when fixing bugs in Python software. Python developers can utilize the study results to deal with dynamic features correctly and safely when fixing bugs. This study also summarizes 14 change patterns for fixing bugs in dynamic feature code, which benefits the developers' bug fixing tasks.

3 Experimental setup and feature changes collection

Since it was proved that some dynamic features would affect the change-proneness of Python files [8], this paper analyzes the changes of Python dynamic feature code, especially when fixing bugs, to reveal the benefits and costs of dynamic features. By combining quantitative and qualitative analyses, our experiments explore the change occurrences and change behaviors of dynamic feature code when developers fix bugs in Python software. Two research questions in this paper are listed as follows.

RQ 1. How do dynamic feature code changes occur in practice?

The goal is to present an initial view about the occurrences of dynamic feature changes during the evolution of Python software. To examine the characteristics of feature change occurrences, we compare the changes of dynamic feature code with non-dynamic feature code when fixing bugs, and also compare the changes of dynamic feature code between bug-fix activities and non-bugfix activities.

RQ 2. How do developers change dynamic feature code when fixing bugs?

We assumed that dynamic features have both positive and negative impacts on Python projects.

Table 2 Descriptive information of analyzed subject projects

Domain	Projects	Avg. of #files	Avg. of #commits	Avg. of #bugs
Data processing	Scipy, Numpy, Matplotlib, Scikit-learn	607	16727	1363
Web	Django, Django-rest-framework, Boto	1014	11630	8849
Media	Beets, Youtube-dl	372	7749	205
Deployment	Ansible, Deis, Fabric	188	9019	1415
Development	Ipython, Sqlmap	359	14109	735
Visualization	Glances, Powerline, Wagtail	220	3209	418

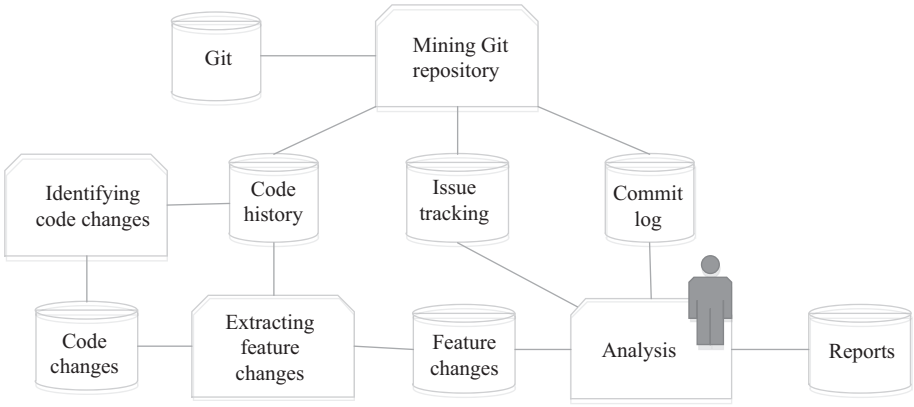

Figure 2 Infrastructure of analyzing feature changes at bug-fix commits.

A qualitative analysis on dynamic feature code changes is carried out to validate this assumption. In particular, we inspected and analyzed practical change occurrences to observe the way in which developers change Python dynamic feature code when fixing bugs.

This section introduces the analyzed projects in this paper and the method of analyzing dynamic feature code changes below.

3.1 Analyzed Python projects

Table 2 describes 17 popular Python projects from GitHub [2)] studied in this paper. These projects cover different domains, including data processing, web, media, deployment, development, and visualization. The datasets were collected in October 2016. We selected these projects following a number of criteria. First, these projects experienced a large number of bugs and feature changes in history. Second, the issue-tracking system in GitHub records all bug issues in the project by naming issue label as "bug" or "bug-report". However, the bug issues in some projects contain unreal bugs such as refactoring issues [12,13]. We selected the projects in which bug issues describe real bugs in the issue-tracking system. Third, the commit messages in these projects record bug-fix description following certain patterns; for example, containing tokens such as "fix" or "patch". Therefore, we can extract bug-fix commits automatically by mining bug-fix description patterns in commit messages. In our corpus, more than 182000 commits and 40000 bugs were collected from 17 subject projects. This paper analyzes the changes of dynamic features at the bug-fix commits of these projects.

3.2 Analysis strategy

Figure 2 shows the infrastructure of analyzing dynamic feature changes. The infrastructure consists of three main components: Mine Git repository, extract code changes, and identify feature changes.

Mine git repository. For each subject project in this study, all code revisions are available on Git, which is a well-known distributed revision control system. We filtered the code revisions that were committed for merging two branches or updating files without Python code. Among the remaining

2) https://github.com.

revisions of subject projects, we differentiated bug-fix commits from non-bugfix commits. One practical method is by mining the issue-tracking system and commit messages to identify the revisions committed for fixing bugs [14–20]. First of all, we extracted bug issues from the issue-tracking system by selecting bug-related label names. Bug issues are identified by issue IDs and are considered convincing bug reports. Afterwards, bug-fix commits are picked out if the commit messages cover fault tokens and real bug issue IDs. Finally, we divided all commits of subject projects into bug-fix commits C_{bf} and non-bugfix commits $C_{\text{non-bf}}$.

Extract code changes. We improved the psydiff [3)] tool to extract source code changes at each commit. psydiff is a source code comparison tool that compares the ASTs of two Python files and visualizes fine-grained code changes between them. We improved the performance of psydiff by reducing the complexity of similarity computation algorithm and avoiding comparing comments in Python files. Afterwards, some neighboring fine-grained changes were connected after analyzing the contexts of changed AST nodes. We grouped the connected changes as a changed code block to avoid too many isolated fine-grained changes. For each commit c, we defined Insert(c), Delete(c), Update(c) to denote the set of all inserted, deleted, modified code blocks or fine-grained code at commit c respectively.

Identify feature changes. In order to identify dynamic feature code, we summarized the formal usages of Python dynamic features (e.g., the built-in and user-defined function names). Dynamic feature code was recognized by mapping AST characteristics to the formal usages of dynamic features. Following this method, the elements in Insert(c), Delete(c), Update(c) were scanned to extract those affected by each dynamic feature df, which are represented as Insert(c, df), Delete(c, df), Update(c, df) respectively. Meanwhile, Change(c, df) denotes the set of changed code blocks or fine-grained code affected by df, which combines Insert(c, df), Delete(c, df), Update(c, df) together.

Following the selected Python projects and the method of analyzing feature changes described above, the next two sections present the manner in which we conducted the experiments and what findings we can discover towards the investigation of two research questions.

4 Occurrences of dynamic feature code changes

This section reports on the investigation and empirical result of RQ1. In order to discover the characteristics of dynamic feature change occurrences in practice, we conducted an empirical study to explore: (1) how many dynamic feature code changes occur for fixing bugs; (2) when fixing bugs, whether dynamic feature code is changed differently with other code; (3) whether certain kinds of feature changes are related with bug-fix activities or non-bugfix activities.

4.1 Occurrences of feature changes when fixing bugs

We identified up to 17926 bug-fix commits in all the subject projects. Table 3 reports the number of bug-fix commits that witnessed the changes of dynamic feature code (including inserts, deletes, updates, and any kind of change). It is worth noting that the fixing activities at one commit may lead to multiple dynamic feature changes.

We find that developers change dynamic feature code widely when fixing bugs in subject projects. In total, feature changes occurred in the fixing activities of 3334 bug-fix commits. Table 3 shows that the change occurrences of different dynamic features during bug-fix activities are uneven. In particular, isinstance, getattr, hasattr, and super are changed more frequently for fixing bugs, involved in 536 to 1014 bug-fix commits. However, some other dynamic features are barely changed during bug-fix activities. For instance, developers changed vars, globals, locals, compile, eval, execfile, and reload code in less than 10 bug-fix commits. The reasons for this are multifold. First, it was reported that some dynamic features in the Introspection category are related to software change-proneness [8]. For instance, visiting an attribute dynamically using getattr tends to be error-prone when the attribute no longer exists, which leads to more changes of getattr code. In fact, getattr changes account for 15% of feature changes during bug-fix

3) https://github.com/yinwang0/psydiff.

Table 3 Number of bug-fix commits that experienced dynamic feature changes in 17 subject projects

Dynamic features	Insert	Delete	Update	Change
hasattr	370	149	261	552
getattr	345	133	420	671
type	65	38	72	151
issubclass	24	10	32	54
isinstance	729	249	499	1014
vars	2	2	1	5
callable	64	25	26	88
globals	2	2	2	5
locals	5	4	2	8
super	484	156	201	536
setattr	100	45	100	180
delattr	9	2	8	18
del	103	68	66	183
compile	3	0	3	6
eval	3	1	2	5
exec	3	0	7	10
execfile	1	0	0	1
__import__	50	32	126	167
reload	3	3	0	5
Total	1994	719	2234	3334

activities. Second, the popular dynamic features experience more changes. For instance, eval and exec are nearly never used by Python developers (occupying only 1% of feature uses). This explains why eval and exec are rarely changed (changed at only 5 and 10 bug-fix commits). Third, certain dynamic features participate in maintenance activities actively. For example, isinstance code is often inserted as a type check to fix wrong type errors (further discussed in the following section); as a consequence, isinstance is frequently changed (occupying 35% of feature changes).

4.2 Dynamic feature changes vs. non-dynamic feature changes

If dynamic features have special effects on bug-fix activities, dynamic feature code is changed differently from the other code when fixing bugs. Therefore, we compared the distribution of three change types between dynamic feature code and non-dynamic feature code during bug-fix activities. For each commit in bug-fix commits C_{bf}, we counted the elements affected by dynamic features in Insert(c), Delete(c), Update(c) to evaluate change types about dynamic features. Similarly, we counted the elements not affected by any dynamic features in Insert(c, df), Delete(c, df), Update(c, df) to evaluate change types about non-dynamic features. Then, we calculated the distribution of three change types of dynamic feature code and non-dynamic feature code separately, which is presented in Figure 3.

Generally, it is more frequent for developers to perform additions on a code element than deletions when fixing bugs [21]. In both kinds of Python code, we can see from Figure 3 that developers insert code in nearly half the cases when they fix bugs in Python programs. When a bug is introduced, the developer may add multiple statements to fix this bug. But in Table 3 we also found that some dynamic feature code is inserted at fewer bug-fix commits than is updated. This is because developers always insert multiple statements at the same commit while they update only a few statements at one commit. When we compared the distribution of change types about dynamic features and non-dynamic features, we found dynamic feature code is inserted and deleted more frequently than non-dynamic feature code. Meanwhile, dynamic feature code is seldom deleted, with such cases occupying only 22% of dynamic feature changes. We can infer that dynamic features are useful in fixing bugs by inserting them into the code. However, when there are bugs around dynamic feature code, developers choose to update it instead of deleting it.

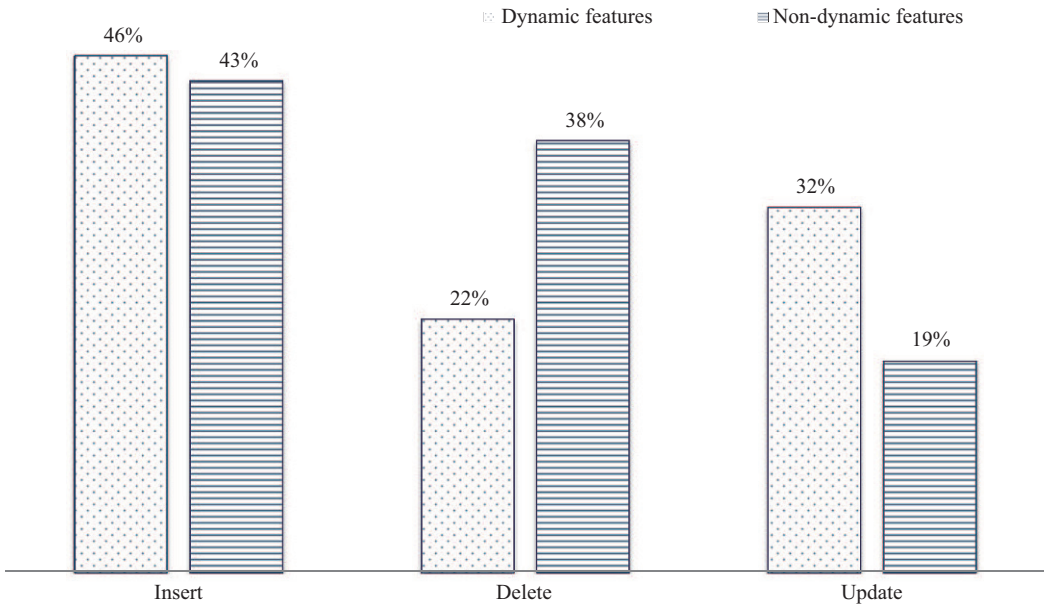

Figure 3 Distribution of three change types of dynamic features and non-dynamic features in fixing bugs.

Table 4 Result of Fisher's exact test which examines the association between feature changes and bug-fix activities

Dynamic features	Insert		Delete		Update		Change	
	p-value	OR	*p*-value	OR	*p*-value	OR	*p*-value	OR
hasattr	**1.28E−24**	**2.079794**	**1.54E−08**	**1.873966**	**2.68E−13**	**1.925916**	**6.58E−26**	**1.958367**
getattr	**7.64E−27**	**2.177587**	**0.000182**	**1.542809**	**1.53E−26**	**2.417415**	**1.34E−28**	**2.059578**
type	**0.014909**	**0.724520**	0.605477	0.900660	0.380095	0.870287	0.099100	0.832316
issubclass	0.194856	1.365870	0.416609	1.359397	**0.015452**	**1.782317**	0.110369	1.398416
isinstance	**3.07E−38**	**1.924559**	**6.78E−07**	**1.534279**	**3.79E−16**	**1.712424**	**1.43E−37**	**1.814357**
callable	**7.03E−07**	**2.301692**	**0.000777**	**2.31738**	0.166888	1.416322	**2.13E−08**	**2.270205**
super	**3.97E−45**	**2.72281**	**1.56E−09**	**2.028205**	**1.03E−18**	**2.653804**	**4.97E−47**	**2.504783**
setattr	**8.76E−15**	**2.865361**	**5.57E−05**	**2.300739**	**5.11E−22**	**5.308334**	**3.92E−14**	**2.517646**
delattr	**5.69E−05**	**7.417112**	0.185077	3.02071	**0.000286**	**10.87703**	**2.66E−05**	**6.232933**
del	0.019203	1.304067	**0.004101**	**1.516179**	**0.004678**	**1.555887**	**0.001716**	**1.347267**
__import__	**2.45E−14**	**4.588762**	**2.52E−05**	**3.358702**	**3.35E−14**	**4.64069**	**2.15E−14**	**3.834873**

4.3 Bug-fix feature changes vs. non-bugfix feature changes

Fisher's exact test [22] is used to examine the association between feature changes and bug-fix activities by comparing change occurrences during bug-fix activities and non-bugfix activities. The test classifies all entities in two ways and examines the significance of the relationship between the two ways of classification. In particular, for each change type (inserts, deletes, and updates) and each dynamic feature, the commits of 17 Python projects were divided into four groups: (1) the bug-fix commits that experience this feature change; (2) the bug-fix commits that do not experience this feature change; (3) the non-bugfix commits that experience this feature change; (4) the non-bugfix commits that do not experience this feature change. Based on the four groups of the commits, Fisher's exact test calculated the *p*-value and the odds ratio (OR) for evaluating the association. If the test result shows that this feature change during bug-fix activities significantly differs from non-bugfix activities (p-value <0.05), we referred to the OR value to examine whether this feature change is related with bug-fix activities (OR >1) or non-bugfix activities (OR <1). To guarantee precision of Fisher's exact test results, we filtered out those dynamic features that are changed at less than 10 bug-fix commits, shown in Table 3: vars, globals, locals, compile, eval, exec, execfile, and reload.

Table 4 presents Fisher's exact test result. The numbers in bold denote that this feature change shows

a significant relationship with bug-fix activities (p-value <0.05 and OR >1). The table reveals that many kinds of dynamic feature changes are significantly related to bug fixing. Generally, all dynamic features are proved more likely to be inserted, deleted, or updated when fixing bugs. Note that the test result does not prove any relationship between any feature changes with non-bugfix activities (p-value <0.05 and OR <1).

We also note some observations from Table 4. First, the changes of reflection dynamic feature code (including isinstance, type, issubclass, and callable) are significantly related to bug fixing. These dynamic features are inserted to check types when fixing wrong type errors. Meanwhile, the parameters in dynamic feature functions may be updated when the checked types of the variables are wrongly set by developers. Second, the significant relationship between setattr, getattr, delattr, and hasattr with bug-fix activities suggests developers' concern about dynamic access of attributes. Owing to the fact that the names of attributes and the structures of objects are changeable after applying setattr or getattr, the incorrect or incomplete changes would lead to attribute errors. In order to fix attribute errors, new dynamic feature code is introduced or the parameters in feature code are modified.

After exploring the characteristics of feature change occurrences through three experiments, we can conclude from the results of RQ1 as follows.

In 17 Python projects, dynamic feature code was changed at 3334 out of 17926 bug-fix commits. On the one hand, dynamic feature code is changed differently from non-dynamic feature code when fixing bugs, with greater probabilities of inserts and updates than deletes. On the other hand, for 11 common dynamic features, the changes of dynamic feature code are significantly related to bug-fix activities rather than non-bugfix activities.

5 Change behaviors of dynamic feature code when fixing bugs

To answer RQ2, this section presents a qualitative analysis on feature changes when fixing bugs. In fact, feature changes at bug-fix commits may be unrelated to bugs. In order to analyze the actual motivation and effects of feature changes, we manually inspected the change behaviors of dynamic features at 226 bug-fix commits. For each change occurrence during bug-fix activities, we analyzed the root cause of the bug and also observed the change behavior for fixing bugs. Generally, we selected and summarized two scenarios when dynamic feature code is changed for fixing bugs: (1) introduced to aid bug fixing; (2) updated or deleted to fix the bugs caused by the misuse of dynamic features.

5.1 Introduced to aid bug fixing

In most cases, dynamic features are changed to cooperate with other core fixing code; for example, inserting super when creating a new subclass. For all the inserts of dynamic feature code during bug-fix activities, we checked whether the inserts are really used for fixing bugs instead of supporting requirements. Finally, we found that 27 out of 75 feature inserts are important in aiding bug fixing in deed. As a part of our results, in Table 5 we list the description and root causes of the bugs, inserted dynamic features, change behaviors for fixing the bugs, and number of occurrences at analyzed bug-fix commits. Two main change behaviors were discovered in aiding bug fixing: add type check, and add attribute check.

Add type check. In our corpus, adding type check is the most frequent change behavior for fixing bugs. Owing to the lack of variable type declarations and the limitations of type inference techniques, Python programs tend to produce type errors [23]. Some reflection functions of Python, including isinstance, issubclass, and type, help check variable types dynamically. It protects the following code, which would be executed with expected types. For instance, consider the following bug-fix example from ansible [4]:

```
−    tmp_dict[ans_opt] = attributes[ssh_opt]
+    attr = attributes[ssh_opt]
```

4) https://github.com/ansible/ansible.

Table 5 Description of the bugs and feature inserts for fixing bugs

Bug	Bug causes	Introduced dynamic features	Change behaviors	#Occurrences
Wrong type	Wrong execution result, invalid input, incompatible execution environment, incompatible library version	isinstance, issubclass, type	Add type check	17
Invalid access of attribute	Never created attribute, deleted attribute, buggy attribute name, or null class/instance object	hasattr, getattr	Add attribute check	8
MRO confliction	Deriving new class wrongly in multiple inheritance	super	Support MRO	2

```
+                    if type(attr) is list:
+                        attr = attr[0]
+                    tmp_dict[ans_opt] = attr
```

Once paramiko library was updated from 1.7.2 to 1.15.2, attributes[ssh_opt] was changed to be a list object. In order to deal with this update, the type statement was inserted to check the type of attributes[ssh_opt], which allows the code to perform the correct operations according to the types of attributes[ssh_opt]. In summary, such dynamic type check code can be inserted to fix wrong type bugs. We discovered 17 bugs that were fixed by introducing isinstance (7 cases), issubclass (3 cases), and type (7 cases) as type checks.

Add attribute check. Another significant change behavior for aiding bug fixing is adding attribute check. In Python, the attributes of Python objects are allowed to be dynamically appended or deleted. The code would raise AttributeError if developers visit an attribute which no longer exists. It usually results from a buggy attribute name, a deleted attribute, a never created attribute, or a null class/instance object. hasattr and getattr are often introduced to check the existence of an attribute when fixing such AttributeError bugs. Consider the example from ansible:

```
-                    if self._role and (value is None or extend):
-                        parent_value = getattr(self._role, attr)
+                    if self._role and (value is None or extend) and hasattr(self._role, attr):
+                        parent_value = getattr(self._role, attr, None)
```

This commit fixes the lookup of parent attribute when parent self._role does not have the attribute attr (bug id: #14100). Therefore, hasattr was introduced to check the existence of the attr attribute to avoid the failure of the getattr function. We found eight different cases for such inserts of dynamic attribute checks that can fix bugs caused by the failure of attribute access.

In Python software that uses multiple inheritance, Method Resolution Order (MRO) specifies the order in which base classes are searched when looking for a method, but MRO confliction forbids the naive programmer from creating ambiguous inheritance hierarchies. In this situation, super is introduced to construct the linearization in deriving a subclass to fix the bug about wrong MRO, which is shown in the last row of Table 5. Extra insert patterns of dynamic features also exist in our corpus. In fact, dynamic features support not only rapid development but also rapid repairs. Generally, when a bug in Python code concerns an unexpected execution status, dynamic features could be introduced to update the execution status to the correct status in a variety of ways, for example, adding dynamic check to avoid harmful operations or modifying the state of Python object. However, for various static programming languages, fixing such an unexpected execution status may need more effort with various code changes.

Table 6 Description of bugs and feature updates or deletes for fixing bugs

Bug	Bug cause	Updated/Deleted dynamic features	Change behaviors	#occurrences
Wrong type check	Supporting too specific types, too generic types, or neighboring types	isinstance, issubclass, type	Change type parameter	20
Dynamic import error	Non-existing module (incompatible library version, wrong module name, absent module), wrong configuration parameter	__import__	Add exception handling, add check before dynamic code, add check after dynamic code, change configuration parameter	14
Invalid dynamic access of attribute	Non-existing attribute (already deleted attribute, non-existing class/instance, wrong attribute name, wrong class/instance name)	getattr	Add exception handling, return default value	10
Invalid dynamic updating of attribute value	Updating the attribute that cannot be overwritten	setattr	Add check before dynamic code	6
Dynamic compile error	Invalid access of code to be compiled, wrong configuration parameter	compile	Add exception handling, add check before dynamic code, change configuration parameter	4
Invalid dynamic deletion of attribute	Non-existing attribute (already deleted attribute, non-existing class/instance, wrong attribute name, wrong class/instance name)	delattr	Add exception handling	4
Invalid dynamic deletion of variable	Non-existing variable (already deleted variable, wrong variable name)	del	Add exception handling	3
Undefined variable error	Typo	instance, super	Change variable name	3
Dynamic code execution error	Invalid expression or statement to be evaluated	eval, exec	Add exception handling, add check before dynamic code, add check after dynamic code	3

5.2 Updated or deleted to fix bugs in dynamic feature code

Although dynamic feature code is introduced to aid fixing bugs in some cases, we also discovered more cases in which dynamic feature code contains bugs and then is fixed by delete or update behaviors. Similarly, we checked whether the updates and deletes are really caused by fixing bugs hidden in dynamic feature code. In particular, we found 67 occurrences of feature deletes and feature updates during bug-fix activities because of really erroneous dynamic feature code. We report the description of the bugs and feature changes in Table 6. As supplementary information, we also abstracted bug patterns, preconditions, and detailed change types in Table 7. Then, we describe several frequent change behaviors when fixing erroneous dynamic feature code.

Change type parameter. One of the most common change behaviors in fixing erroneous dynamic code is changing parameters in dynamic feature function calls, especially in isinstance, issubclass, and type statements. Developers use isinstance, issubclass, and type to ensure that the program accepts the variables with given types. However, when the type parameters in these type-checking conditional statements are wrong and produce failures, the dynamic feature code would be updated by giving another group of types. Generally, such bugs often occur when developers use isinstance, issubclass, and type to support too specific types, too generic types, or neighboring types. An example in ansible is as follows:

Table 7 Change patterns in fixing erroneous dynamic feature code

Bug pattern	Precondition	Change type
isinstance(object, typeinfo)	Wrong type check of object	Parameter update: typeinfo
issubclass(object, typeinfo)	Wrong type check of object	Parameter update: typeinfo
type(object) == typeinfo	Wrong type check of object	Parameter update: typeinfo
__import__(name[, globals[, locals[, fromlist[, level]]]])	Non-existing module: name	Statement insert: try-except statement insert: if
__import__(name[, globals[, locals[, fromlist[, level]]]])	Wrong configuration	Parameter update: globals, locals, fromlist, level
getattr(object, name)	Non-existing attribute: name	Statement insert: try-except parameter insert: Default-Value
setattr(object, name, value)	Unallowed to overwrite: name	Statement insert: if
compile(source, filename, mode[, flags[, dont_inherit[, optimize]]])	Invalid access of source	Statement insert: try-except
compile(source, filename, mode[, flags[, dont_inherit[, optimize]]])	Wrong type of source	Statement insert: if
compile(source, filename, mode[, flags[, dont_inherit[, optimize]]])	Wrong configuration	Parameter update: mode, flags, dont_inherit, optimize
eval(expression)	Invalid expression: expression	Statement insert: try-except statement insert: if
exec(object)	Invalid statement: object	Statement insert: try-except statement insert: if
del name	Non-existing variable: name	Statement insert: try-except
delattr(object, name)	Non-existing attribute: name	Statement insert: try-except

```
−        if isinstance(log_args[arg], unicode):
−            msg = msg + arg + '=' + log_args[arg] + ' '
−        else:
−            msg = msg + arg + '=' + str(log_args[arg]) + ' '
+        if isinstance(log_args[arg], basestring):
+            msg = msg + arg + '=' + log_args[arg].decode('utf-8')
+        else:
+            msg = msg + arg + '=' + str(log_args[arg]) + ' '
```

Developers used the instance to ensure that the type of log_args[arg] is unicode, but omitted that log_args[arg] also accepts str. This bug would result in the execution of a wrong branch. In order to fix this bug, developers changed the type parameter from unicode to basestring (the base class of str and unicode). This update of isinstance allows the execution of this code to always follow a correct branch. It suggests that developers should take care when they use type, isinstance, and issubclass considering the implementation of inheritance.

Add a check & add an exception handling. Conditional statements are inserted or updated to check beforehand or afterward, whether the execution of some dynamic feature code would be dangerous or not. This change behavior fixes the incorrect or absent control condition. For instance, it would cause a failure when developers use setattr to update the attribute that cannot be overwritten, thus a conditional statement is added to check whether this attribute is allowed to be overwritten. An alternative solution is adding the exception handling for it when the execution of dynamic feature code may throw an exception. In our corpus, most potentially harmful dynamic code can hardly be replaced with a safer form of another code. These bugs include invalid dynamic deletion of variables (del) or attributes (delattr). When deleting a variable or attribute raises an exception at runtime, developers can just ignore the exception by adding the exception handling because the variable or attribute to be deleted has been non-existent already. A similar bug occurs when developers dynamically import a module by using __import__ but the module is absent. This kind of bugs can hardly be prevented by conditional checks, but it is advisable to add the exception handling for __import__ code to provide an alternative import or

a detailed error message.

We can infer from Tables 6 and 7 that the bugs in dynamic feature code are often caused by the misuse of dynamic features. As dynamic features support rapid development of Python software, developers often enjoy the strong function of dynamic features arbitrarily but ignore the potential dangers they pose. Some bugs are related to developers' confusion about dynamic types of Python variables, especially the bugs in isinstance, type, and issubclass statements. These bugs can be fixed by modifying the type parameters. Many other bugs caused by the misuse of dynamic features including __import__ and del can be fixed by adding a check or adding an exception handling.

After inspecting and analyzing various change occurrences of dynamic feature code during bug-fix activities, we have discovered several general patterns that appeared when introducing dynamic features to fix bugs and when changing dynamic feature code to fix the bugs caused by the misuse of dynamic features. In summary, we can draw some conclusion from the results of feature change behaviors when fixing bugs for RQ2.

Generally, developers change dynamic feature code to fix bugs in two scenarios. First, they often insert dynamic feature code as type check or attribute check to support bug-fix activities. Second, they update or delete dynamic feature code to fix bugs in dynamic feature code, often by adding a check or adding an exception handling. In our corpus, the latter scenario is more common than the former, which reveals that dynamic feature code tends to be error-prone.

6 Discussion

From the results presented in Sections 4 and 5, this section discusses the implications for developers and researchers, application for automatic program repair, and limitations of this study.

6.1 Implications for developers and researchers

Overall, the results indicate both benefits and costs of Pythons dynamic features in software maintenance, particularly during bug-fix activities. We list the implications of the empirical findings below.

When developers fix bugs in dynamic feature code, they tend to update it instead of deleting it. According to the result of RQ2, developers would delete or update dynamic feature code to fix the bugs caused by the misuse of dynamic features. Furthermore, we found that dynamic feature code is more likely to be updated than to be deleted when fixing bugs compared with other code from Figure 3. In other words, when dynamic feature code contains a bug, developers choose to update it instead of replacing it with other code. There may be two reasons behind it. First, the bugs can easily be fixed by updating dynamic feature code, following the fixing patterns presented in Table 7. Second, dynamic feature code can hardly be replaced by other static code. Dynamic features allow developers to check the object state or modify the execution status at runtime, which cannot be realized by static code.

Type errors and attribute errors are common bugs in Python programs. After investigating the manner in which developers change feature code when fixing bugs, type errors and attribute errors were discovered in both bug-fix scenarios. Owing to the lack of declarations of variable types and allowing dynamic changes of class/instance structures in Python, type errors and attribute errors are common in Python programs. Xu et al. [23] also verified that attribute errors, type errors, and unicode errors are very common in Python applications. isinstance, issubclass and type are often introduced to aid fixing bugs, but they also cause wrong type check bugs. In fact, we found that isinstance, issubclass, and type were inserted to fix type error bugs in 16 cases, which means that these dynamic features benefit bug-fix activities. However, we also found 18 delete and update occurrences to fix bugs contained by them. Dynamic type check is essential in guaranteeing type safety in Python software that lacks type declarations, but misuse issues become the new threats to software quality. Similarly, hasattr and getattr are inserted to add attribute check for aiding fixing, but they also produce bugs when wrongly accessing or updating attributes. Therefore, isinstance, issubclass, type and hasattr, getattr are useful for fixing

type errors and attribute errors respectively, but the usages of these dynamic features in Python programs are also error-prone. It suggests that developers should introduce these two groups of dynamic features carefully.

Dynamic feature code is often changed following particular patterns during bug-fix activities. From the results of RQ1, we can see that the changes of dynamic features have significant effects on bug-fix activities in Python software. Furthermore, the investigation of RQ2 reveals that some feature changes follow particular patterns when fixing bugs. To aid bug fixing, dynamic features are often introduced as type checks or attribute checks. Meanwhile, to fix bugs in dynamic feature code, dynamic features are often changed by adding a check or adding an exception handling. This implication motivates the study of automatic program repair after learning historical change behaviors of dynamic features during bug-fix activities. A more detailed approach is discussed in the next subsection.

6.2 Recommending change behaviors for fixing bugs in dynamic feature Code

The results of this study provide insights into automatic program repair with dynamic features. Particularly, fixing patterns with dynamic features can be abstracted by learning bug patterns and change behaviors of dynamic feature code performed in fixing history of Python software, including two scenarios discovered in the investigation of RQ2: introducing dynamic features to aid fixing bugs, and deleting or updating dynamic feature code to fix bugs in dynamic feature code. In the first scenario, as dynamic features are introduced to fix bugs in a variety of ways, it is difficult to summarize prevalent bug patterns. In the second scenario, the results in Tables 6 and 7 reveal prevalent bug patterns in dynamic feature code and it is more instructive to detect and fix erroneous dynamic feature code. After developers locate bugs in dynamic feature code, the results of this study provide opportunities to recommend change behaviors of dynamic features to fix bugs based on code characteristics and bug patterns in our corpus. In the following, we propose an approach of recommending change behaviors for fixing erroneous dynamic feature code, which benefits from the results of this study.

First of all, we assume that similar bug code is fixed with similar behaviors by developers, because there are always similar or even duplicated bugs in bug reports and developers tend to fix them with the same patches. The key idea of recommending change behaviors for fixing the bugs caused by the misuse of dynamic features is searching fixing patterns of similar erroneous dynamic feature code by analyzing and comparing code characteristics and bug patterns. The recommendation is processed as follows.

Abstracting change behaviors. We summarized bug patterns in erroneous dynamic feature code and abstracted change behaviors when fixing bugs to obtain change patterns. A portion of the change patterns abstracted from our corpus are presented in Table 7. In our abstraction, a change pattern is made up of the following components: the bug pattern, precondition, and change type. The bug pattern is the usage of a dynamic feature and the precondition describes the context in which the bug appears. The change type is computed by the source code differencing tool and reveals the way in which the bug is fixed at the commit.

Computing code similarity. When a bug is found in dynamic feature code, we search similar erroneous dynamic feature code in the fixing history and rank them by similarities. The similarity between two snippets of dynamic feature code can be quantified by three factors: the types of dynamic features, similarity of code patterns, and similarity of ASTs.

Recommending ranked change patterns. The change patterns of similar erroneous dynamic feature code (e.g., insert if or try-except) are recommended to users to fix a dynamic feature bug, with the rank of code similarity. If no similar erroneous code is found in history, we recommend fixing patterns according to the prevalence of change patterns in our corpus.

6.3 Threats to validity

Several threats to validity in this paper are listed as follows.

Threats to construct validity. There exist no perfect ways in which to collect bug-fix commits from Git. We identified bug-fix commits of Python projects by mining bug IDs and fault keywords in

commit messages and matching bug IDs with bug issues. The question is, it is difficult to recognize bug-fix actions accurately from casual commit messages. The studied Python projects in this paper are mature and large-size projects and we can retrieve formal commit messages in these projects. The second problem lies in the recognition of dynamic feature code. The assumption is that dynamic features in Python are always applied by calling standard APIs. However, some non-standard ways are also used to trigger Python dynamic features. For instance, if one dynamic feature is encapsulated in a user-defined method, then the invocations of this method are missed when identifying dynamic feature code. One alternative is analyzing Python code dynamically, but it seems impractical to perform dynamic analysis on large-scale projects. In fact, non-standard ways are rarely used to trigger Python dynamic features.

Threats to internal validity. We manually inspected, explained, and classified change occurrences at 226 bug-fix commits. However, it is essentially subjective and application-specific to understand the change of a given dynamic feature. We intend to invite skilled developers to fix or confirm our conclusion in future. In addition, new change patterns may occur in other change occurrences that were not inspected in our study. Although manual inspection is time-consuming, we plan to increase the database and analyze more feature changes. Internal validity threats also include inherent deficiencies from the psydiff tool. We utilized psydiff to extract fine-grained code changes between two versions of source code. Although we have improved psydiff by enhancing similarity computation algorithm and grouping related fine-grained changes, the tool still exhibits inherent limitations. For example, it would be unable to map two unchanged statements if they are surrounded with consecutive fine-grained changes.

Threats to external validity. Our study results are threatened by the bias of the datasets. In this paper, we selected a corpus of 17 popular Python software projects from GitHub, but we cannot generalize the results of this study to commercial Python software. We could replicate the study on a wider variety of Python software to obtain more general results. Furthermore, we concentrated on the benefits and costs of dynamic features of the Python language. Python is a popular dynamic language that carries many types of dynamic features. We assume that the dynamic features are commonly used and changed in Python projects. Our study results need to be checked by performing the same study on other dynamic languages such as JavaScript, Ruby, and PHP.

7 Related work

Dynamic feature uses. The uses of dynamic features have been addressed by researchers in many programming languages. Callaú et al. [6] explored the motivation and behaviors of dynamic feature uses in Smalltalk. They concluded that dynamic features are rarely used in Smalltalk. Their results also suggested that most dynamic features could be replaced with more static code. Richards et al. [5] studied the behaviors of JavaScript dynamic features to analyze the way in which dynamic features are used by developers in JavaScript code. They demonstrated that some dynamic features, including eval and deletions, are used frequently. In their later work [4], they performed a further analysis on the usages of eval in Javascript programs. They discovered some cases where eval is misused by developers and they suggested that eval could be replaced with a safer form of code. Their findings of dynamic feature misuses reveal the costs of dynamic features in software maintenance. However, these studies only provide an initial view on the usages of dynamic features.

Python dynamic features. In recent years, researchers began to analyze the dynamic features in Python code. As a typical dynamic language, previous studies on Python language mainly include program slicing techniques [24,25], information flow analysis [26], type analysis [23,27–29], polymorphism analysis [30], and alias analysis [31]. The first study on Python dynamic features was conducted by Holkner and Harland [2], which traced the usages of dynamic features in Python projects. The trace results show that dynamic feature activities mainly occur at startup phases of Python projects. However, in 2014, Akerblom et al. [1] carried out a similar study but demonstrated the behaviors of Python dynamic features neither prominently occur during program startup nor are buried in the library. Concerning the negative aspects of dynamic features, Wang et al. [8] explored the relationships between dynamic features

and the changes of Python code. Their results show that the files with the participation of dynamic features are more change-prone. Their findings inspired this study on analyzing the benefits and costs of dynamic features. In 2016, Lin et al. [32] developed an automatic tool to extract fine-grained code change types for Python code and investigated the characteristics of Python code changes. They also collected the fine-grained change types of dynamic feature code in history. They found that isinstance, type, hasattr, and getattr are the most frequently changed dynamic features and update is the most common change type of dynamic features, which is consistent with our results. This paper extended their work by further analyzing feature changes during bug-fix activities and exploring the benefits and costs of dynamic features.

Fine-grained code changes. Code changes reveal the features and trends of given constructs and are broadly treated as an essential element of software evolution. Many differencing approaches and tools have been proposed and developed to extract the changes between two files, including AST-based differencing approaches [32–35], semantic-based differencing approaches [36–38] and line-based textual differencing approaches [39]. In this paper, an AST-based differencing technique was improved to analyze feature code changes in Python. The extracted code changes occurred in history reveal past, current, or future states of the projects [40, 41]. In order to collect code changes, Omori and Maruyama [42] implemented an Eclipse plug-in for recording and collecting editing operations that have ever been applied to the target source. They retrieved editing operations from Eclipse undo history and some retrieved operations were grouped according to their spatial distance. In order to determine the motivation of a code change, Rastkar and Murphy [43] believed the change motivation can be inferred by piecing together document information from relevant projects. They used multi-document summarization techniques to generate a description that concisely explains why code changed. In addition, Canfora et al. [44] carried out an exploratory study on code changes during the histories of four systems to analyze the interrelations between code change entropy with four factors, such as refactoring activities. These studies on code changes facilitate the understanding of software evolution.

Bug-inducing and bug-fixing code changes. Following the research on mining and analyzing code changes, code changes have been widely used for defect prediction [45] and bug localization [46]. Kamei et al. [45] built the models to predict whether or not a change would result in a defect. They pointed out that defect prediction at the fine-grained change level can save significant effort over coarse-grained predictions. Wen et al. [46] discovered that most of the bugs are introduced by changes and bug-inducing changes can help developers in debugging. According to this finding, they improved the performance of IR-based bug localization by leveraging fine-grained code change histories. Similar to our work, there are some previous researches that also attempted to study fine-grained code changes during bug-fix activities [8, 47, 48]. Pan et al. [48] defined 27 bug fix patterns from the repositories of seven open source projects and found the most common patterns are Method Call. What is different, are that the most common patterns in dynamic feature code turn out to be adding a check or adding an exceptional handling. They also conducted a correlation analysis on the extracted patterns on the seven projects and discovered six have very similar bug fix pattern frequencies. A similar study was explored by Zhao et al. [47] who developed an automatic classification tool to categorize five change types and nine change subtypes in bug fixing code. They obtained a similar finding that the bug fix pattern frequencies tended to be similar across different projects. This finding supports our idea of recommending bug-fix changes in dynamic feature code based on the summarized change patterns in Table 7.

8 Conclusion

Dynamic features play an essential part in software development, but they also pose threats to code quality. Consequently, dynamic features have complex effects on software maintenance. This paper presents a comprehensive study to analyze the changes of dynamic feature code when fixing bugs. We aim at assessing the number of dynamic feature code changes occurring during software evolution, and the way developers change dynamic feature code when fixing bugs. The results show that: (1) the changes of

dynamic feature code are significantly related to bug-fix activities rather than non-bugfix activities; (2) compared with non-dynamic feature code, dynamic feature code is inserted or updated more frequently when fixing bugs; (3) developers often insert dynamic feature code as type checks or attribute checks to fix type errors and attribute errors; (4) the misuse of dynamic features introduces bugs in dynamic feature code and the bugs are often fixed by adding a check or adding an exception handling.

As a benefit of this study, we gain insights into the manner in which developers and researchers cope with dynamic features during bug-fix activities. In future, we intend to extend our work in two directions. First, we plan to study the changes of dynamic features during other maintenance activities such as refactoring. Furthermore, inspired by these findings, we plan to develop a tool for automatic program repair or automatic defect detection using the results of this study. For example, we intend to evaluate the approach of recommending change behaviors for fixing bugs in dynamic feature code, which is discussed in Section 6.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61472175, 61472178, 61403187), Natural Science Foundation of Jiangsu Province of China (Grant No. BK20140611), and National Key Basic Research and Development Program of China (Grant No. 2014CB340702).

References

1 Akerblom B, Stendahl J, Tumlin M, et al. Tracing dynamic features in Python programs. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, 2014. 292–295
2 Holkner A, Harland J. Evaluating the dynamic behaviour of Python applications. In: Proceedings of the 32nd Australasian Conference on Computer Science, Wellington, 2009. 19–28
3 Bodden E, Sewe A, Sinschek J, et al. Taming reflection: aiding static analysis in the presence of reflection and custom class loaders categories and subject descriptors. In: Proceedings of the 33rd International Conference on Software Engineering, Waikiki, 2011. 241–250
4 Richards G, Hammer C, Burg B. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In: Proceedings of the 25th European Conference on Object-oriented Programming, Lancaster, 2011. 52–78
5 Richards G, Lebresne S, Burg B, et al. An analysis of the dynamic behavior of JavaScript programs. ACM SIGPLAN Notices, 2010, 45: 1–12
6 Callaú O, Robbes R, Tanter E, et al. How developers use the dynamic features of programming languages: the case of Smalltalk. In: Proceedings of the 8th Working Conference on Mining Software Repositories, Waikiki, 2011. 23–32
7 Dufour B, Goard C, Hendren L, et al. Measuring the dynamic behaviour of AspectJ programs. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, 2004. 150–169
8 Wang B B, Chen L, Ma W W Y, et al. An empirical study on the impact of Python dynamic features on change-proneness. In: Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, 2015. 134–139
9 Park J, Lim I, Ryu S. Battles with false positives in static analysis of JavaScript web applications in the wild. In: Proceedings of the 38th International Conference on Software Engineering Companion, Austin, 2016. 61–70
10 Sanner M F. Python: a programming language for software integration and development. J Mol Graph Model, 1999, 17: 57–61
11 Chen Z F, Ma W W Y, Lin W, et al. Tracking down dynamic feature code changes against Python software evolution. In: Proceedings of the 3rd International Conference on Trustworthy Systems and Their Applications, Wuhan, 2016. 54–63
12 Qian J, Chen L, Xu B W. Finding shrink critical section refactoring opportunities for the evolution of concurrent code in trustworthy software. Sci China Inf Sci, 2013, 56: 012106
13 Chen L, Qian J, Zhou Y M, et al. Identifying extract class refactoring opportunities for internetware. Sci China Inf Sci, 2014, 57: 072103
14 Feng Y, Liu Q, Dou M Y, et al. Mubug: a mobile service for rapid bug tracking. Sci China Inf Sci, 2016, 59: 013101
15 Zhang J, Wang X, Hao D, et al. A survey on bug-report analysis. Sci China Inf Sci, 2015, 58: 021101
16 Chen L, Ma W W Y, Zhou Y M, et al. Empirical analysis of network measures for predicting high severity software faults. Sci China Inf Sci, 2016, 59: 122901
17 Kim S, Zimmermann T, Whitehead E J. Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, 2007. 489–498
18 Fischer M, Pinzger M, Gall H. Populating a release history database from version control and bug tracking systems. In: Proceedings of the International Conference on Software Maintenance, Amsterdam, 2003. 23–32
19 Hall T. Some code smells have a significant but small effect on faults. ACM Trans Softw Eng Methodol, 2014, 23: 33
20 Khomh F, Penta M D, Guéhéneuc Y G, et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empir Softw Eng, 2012, 17: 243–275

21 Zhong H, Su Z D. An empirical study on real bug fixes. In: Proceedings of the 37th International Conference on Software Engineering, Florence, 2015. 913–923

22 Monographs B. Statistical methods for research workers. In: Breakthroughs in Statistics. Berlin: Springer-Verlag, 1992. 66–70

23 Xu Z G, Liu P, Zhang X Y, et al. Python predictive analysis for bug detection. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 121–132

24 Chen Z F, Chen L, Zhou Y M, et al. Dynamic slicing of Python programs. In: Proceedings of the 38th Computer Software and Applications Conference, Vasteras, 2014. 219–228

25 Xu Z G, Qian J, Chen L, et al. Static slicing for Python first-class objects. In: Proceedings of the 13th International Conference on Quality Software, Najing, 2013. 117–124

26 Chen Z F, Chen L, Xu B W. Hybrid information flow analysis for Python bytecode. In: Proceedings of the 11th Web Information System and Application Conference, Tianjin, 2014. 95–100

27 Chen L, Xu B W, Zhou T L, et al. A constraint based bug checking approach for Python. In: Proceedings of the 33rd Computer Software and Applications Conference, Seattle, 2009. 306–311

28 Vitousek M M, Kent A M, Siek J G, et al. Design and evaluation of gradual typing for Python. In: Proceedings of the 10th ACM Symposium on Dynamic Languages, Portland, 2014. 45–56

29 Xu Z G, Zhang X Y, Chen L, et al. Python probabilistic type inference with natural language support. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 607–618

30 Akerblom B, Wrigstad T. Measuring polymorphism in Python programs. In: Proceedings of the 11th Symposium on Dynamic Languages, Pittsburgh, 2015. 114–128

31 Gorbovitski M, Stoller S D. Alias analysis for optimization of dynamic languages. In: Proceedings of the 6th Symposium on Dynamic Languages, Reno/Tahoe, 2010. 27–42

32 Lin W, Chen Z F, Ma W W Y, et al. An empirical study on the characteristics of Python fine-grained source code change types. In: Proceedings of the 32nd International Conference on Software Maintenance and Evolution, Raleigh, 2016. 188–199

33 Member S. Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans Softw Eng, 2007, 33: 725–743

34 Neamtiu I, Foster J S, Hicks M. Understanding source code evolution using abstract syntax tree matching. In: Proceedings of the 2005 International Workshop on Mining Software Repositories, St. Louis, 2005. 1–5

35 Sager T, Bernstein A, Pinzger M, et al. Detecting similar Java classes using tree algorithms. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, Shanghai, 2006. 65–71

36 Apiwattanapong T, Orso A, Harrold M J. A differencing algorithm for object-oriented programs. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, Linz, 2004. 2–13

37 Howitz S. Identifying the semantic and textual differences between two versions of a program. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, White Plains, 1990. 234–245

38 Raghavan S, Rohana R, Leon D, et al. Dex: a semantic-graph differencing tool for studying changes in large code bases. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, 2004. 188–197

39 Kim M, Notkin D. Program element matching for multi-version program analyses. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, Shanghai, 2006. 58–64

40 Purushothaman R, Perry D E. Toward understanding the rhetoric of small source code changes. IEEE Trans Softw Eng, 2005, 31: 511–526

41 Voinea L, Telea A. CVSscan: visualization of code evolution. In: Proceedings of the 2005 ACM Symposium on Software Visualization, St. Louis, 2005. 47–56

42 Omori T, Maruyama K. A change-aware development environment by recording editing operations of source code. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, Leipzig, 2008. 31–34

43 Rastkar S, Murphy G C. Why did this code change? In: Proceedings of the 2013 International Conference on Software Engineering, San Francisco, 2013. 1193–1196

44 Canfora G, Cerulo L, Cimitile M, et al. How changes affect software entropy: an empirical study. Empir Softw Eng, 2014, 19: 1–38

45 Kamei Y, Shihab E, Adams B, et al. A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng, 2013, 39: 757–773

46 Wen M, Wu R X, Cheung S C. Locus: locating bugs from software changes. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 2016. 262–273

47 Zhao Y Y, Leung H, Yang Y B, et al. Towards an understanding of change types in bug fixing code. Inform Softw Tech, 2017, 86: 37–53

48 Pan K, Kim S, Whitehead E J Jr. Toward an understanding of bug fix patterns. Empir Softw Eng, 2009, 14: 286–315