International Journal of Software Engineering
and Knowledge Engineering
Vol. 32, No. 5 (2022) 745–768

DOI: 10.1142/S0218194022500243

An Empirical Study on the Impact of Python Dynamic Typing on the Project Maintenance

Xinmeng Xia*, Yanyan Yan†, Xincheng He‡, Di Wu§, Lei Xu¶
and Baowen Xu‖

State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, P. R. China
**xiaxm@smail.nju.edu.cn*
†*MG1832009@smail.nju.edu.cn*
‡*xinchenghe2016@gmail.com*
§*htuwudi@sina.com*
¶*xlei@nju.edu.cn*
‖*bwxu@nju.edu.cn*

Received 6 December 2021
Revised 16 February 2022
Accepted 9 March 2022
Published 25 May 2022

Python is a popular typical dynamic programming language. In Python, dynamic typing is one of the most critical dynamic features. The lack of type information is likely to hinder the maintenance of Python projects. However, existing work has seldom focused on studying the impact of Python dynamic typing on project maintenance. This paper focuses on the two most common practices of Python dynamic typing, i.e. inconsistent-type assignments (ITA) and inconsistent variable types (IVT). Two approaches are proposed to identify ITA and IVT, i.e. identifying ITA by analyzing Abstract Syntax Trees and comparing identifiers types and identifying IVT by constructing a type dependency graph. In empirical experiments, we first locate the usage of ITA and IVT in 10 open-source Python projects. Then, we investigate the relations between the occurrence of ITA and IVT and the results of maintenance tasks. The study results show that projects are more prone to change as the number of dynamic typing identifiers increases. There is a weak connection between change-proneness and variable dynamic typing. There is a high probability that maintenance time and the acceptance of commits decrease as dynamic typing identifiers increase in projects. These results implicate that dynamic and static variables should be divided while developing new programming languages. Dynamic typing identifiers may not be the direct root causes for most software bugs. The categories of these bugs are worth exploring.

Keywords: Dynamic typing; software maintenance; empirical study.

1. Introduction

Dynamic typing is one of the most critical dynamic features in dynamic programming languages [1]. Variables can be dynamically bound to different types at run-time.

¶Corresponding author.

This feature makes dynamic languages easy to write but hard to read since no typing information is provided for developers to understand variables. Richards *et al.* [2] conducted an empirical study of dynamic behaviors of JavaScript programs to improve the correctness, security, and performance of JavaScript applications. They analyzed how and why the dynamic features are used in JavaScript programs. Hanenberg *et al.* [3] conducted an empirical study on the impact of static typing on Java software maintenance. They investigated whether static type systems improve the maintainability of software systems in terms of understanding undocumented code, fixing type errors, and fixing semantic errors. The results showed that static types are beneficial to understanding undocumented code and fixing type errors.

Python is a typical popular dynamically-typed programming language. Because of the concise coding style and strong support for rapid development, Python has been widely employed to develop various applications, such as Pytorch [4], Django [5], Ansible [6]. In the latest 2021 IEEE Spectrum rankings [7], it is the most popular programming language, ranking No. 1 among all programming languages. Thus, it is also necessary to know the impact of dynamic behaviors in real Python projects. Holkner and Harland [8] evaluated the dynamic behaviors of Python applications. They measure dynamic behaviors on 24 production-stage open-source Python projects by dynamically analyzing the execution of these programs. However, they do not consider the non-executed parts, e.g. unexecuted branches in control structures, that are related to dynamic typing. Chen *et al.* [9] conducted an empirical study on six types of dynamic typing-related practices in nine Python systems. They investigate whether dynamic typing-related practices are connected with software bugs and find that dynamic typing-related practices have a significant positive correlation with bugs occurring.

In this paper, we conduct an empirical study to further investigate the impact of dynamic typing on software maintenance. More specifically, we investigate the impact of two categories of dynamic typing: *Inconsistent Type Assignment* (*ITA* for short) and *Inconsistent Variable Type* (*IVT* for short). *ITA* refers to a variable reassigned with an inconsistent type. *IVT* refers to a variable with uncertain types at a point due to the program executed in different branches. We statically identify *ITA* by Abstract Syntax Tree (AST) analysis and variable types comparison. We propose the type dependency graph to identify *IVT*. After identifying dynamic typing, six research questions are designed to figure out relations between dynamic typing and software maintenance. More specifically, we first count the frequency and distribution of dynamic typing. Then, we separately investigate the relations between dynamic typing and change-proneness, maintenance time, size of commit files, and the acceptance of commits.

To perform the empirical study, we collect 10 popular open-source Python projects from GitHub. These 10 projects include source code, issues, pull requests, and commits. All these 10 projects own more than 10k stars, and they cover different areas such as machine learning and web communication. Our research results show that there is a weak connection between change-proneness and variable dynamic

typing. The usage of dynamic typing can decrease the acceptance of commit files but save software maintenance time. From these results, we obtain implications. For example, dynamic typing is beneficial but dangerous to software maintenance. Researchers and developers may annotate or classify dynamic and static variables while using or developing new programming languages.

In summary, the main contributions of this paper are as follows:

- We conduct an empirical study to investigate the relations between Python dynamic typing and project maintenance. Based on the findings, we identify a set of actionable recommendations for Python software engineers.
- Seven findings are presented to explain the relations between dynamic typing and software maintenance. We observe that projects are more prone to change as the number of reassignment dynamic typing rises. However, there is no connection between change-proneness and variable dynamic typing. There is a high probability that maintenance time decreases as dynamic typing identifiers increase in projects. There is a high probability that dynamic typing reduces the acceptance of commits.
- Implications are given for developers and researchers. Dynamic typing is actually beneficial but dangerous to software maintenance. It may be a good idea to separate dynamic variables and static variables. Researchers may be interested in the categories of bugs caused by dynamic typing in software maintenance.

The rest of this paper is structured as follows. Section 2 discusses our background in dynamic typing and research methods to identify dynamic typing. Section 3 gives our study design, including the experimental environment, research questions, and empirical analysis methods. Empirical results are shown in Sec. 4. Section 5 discusses our research, lists threats to validity, and gives implications. In Sec. 6, we present related work. Finally, we conclude our work in Sec. 7.

2. Dynamic Typing in Python

This section introduces the background of dynamic typing and our proposed approaches to identify dynamic typing identifiers.

2.1. *Background: Dynamic typing*

In Python, all objects can be dynamically created, destroyed, passed as arguments to functions. The dynamic typing enables programmers to bind names to different objects with another type. For example, programmers can have a variable that holds a *String* and later assign a *List* to it at run-time. This kind of dynamic feature may introduce misbehavior in a program. However, compile-time warnings are not reported when operations are valid until incorrect behavior shows up due to wrong types.

Existing work has been presented to track the inconsistent types and provide warnings for dynamic programming languages (e.g. JavaScript [10]). Unlike JavaScript,

Python has stricter typing rules at compile-time, making errors more likely to happen during compilation. For example, '+' can join a string and a number in JavaScript while not allowed in Python. This paper conducts an empirical study to investigate dynamic typing in Python project maintenance. We focus on the two most common but potentially risky usages of Python dynamic typing practices [9].

ITA refers to a variable reassigned with an inconsistent type. Figure 1 presents an example of *ITA*, which is a code snippet from project *pandas*.[a] *pandas* is a package used to provide flexible data structures for data analysis. This is part of a test in pandas to read multiple sheets from a runtime-created Excel file. As shown in Fig. 1, the type of variable dfs is *List* at line 145, while it is reassigned with the type *Dict* at line 146. Thus, dfs is an *ITA* variable.

```
144 ...
145 dfs = [tdf(s) for s in sheets]
146 dfs = dict(zip(sheets, dfs))
147 ...
```

Fig. 1. An example of *ITA* from project "pandas".

IVT refers to a variable with uncertain types at a point due to the program executed in different branches, e.g. possibly *int* or *String* decided by program executions. Figure 2 provides an example of *IVT*, a code snippet of *Pipenv*.[b] *Pipenv* is a tool aiming to provide all packages to the Python user. This piece of code is to return hashes as a pip. As shown in Fig. 2, two methods (i.e. *get_hashes_as_pip* and

```
2441 ...
2442 def get_hashes_as_pip(self, as_list=False):
2443 # type: (bool) -> Union[STRING_TYPE, List[STRING_TYPE]]
2444     hashes = ""
2445     if as_list:
2446         hashes = []
2447         if self.hashes:
2448             hashes = [HASH_STRING.format(h) for h in self.hashes]
2449     else:
2450         hashes = ""
2451         if self.hashes:
2452             hashes = "".join([HASH_STRING.format(h) for h in self.hashes])
2453     return hashes
2454
2455 @property
2456 def hashes_as_pip(self):
2457     # type: () -> STRING_TYPE
2458     hashes = self.get_hashes_as_pip()
2459     assert isinstance(hashes, six.string_types)
2460     return hashes
2461 ...
```

Fig. 2. An example of *IVT* from project "Pipenv".

[a] https://github.com/pandas-dev/pandas/blob/master/pandas/tests/io/excel/test_writers.py.
[b] https://github.com/pypa/pipenv/blob/master/pipenv/vendor/requirementslib/models/requirements.py.

hashes_as_pip) aim to get hashes. Method *get_hashes_as_pip* returns variable *hashes* (line 2453) with type *String* or *List* due to the uncertainty of program executions in conditional branches. For method *hashes_as_pip*, the type of *hashes* (line 2458) is decided by the return value of *hashes_as_pip*, including two possibilities (e.g. *List*, *String*). Hence, *hashes* at line 2458 is an *IVT* variable.

2.2. *Identification of dynamic typing*

This section demonstrates two proposed approaches to identify dynamic typing identifiers: identifying *ITA* by analyzing ASTs and identifying *IVT* by constructing a type dependency graph.

2.2.1. *Identification of ITA*

We identify *ITA* by analyzing AST and comparing variable types. For a given program, the first thing is to parse this piece of code and turn it into AST. Then, we traverse the AST and divide the AST based on different scopes, e.g. module, function, class. For instance, in AST, we locate all function nodes and their subtree. Then, this AST is divided based on function scope. After splitting the scopes, we traverse all child nodes for each subtree in each scope, respectively. If two nodes, in a traverse of a subtree, are initialized variables with the same name in assignment statements, we compare their types and decide whether they are identical. Their types are obtained from the type inference tool. If the types are identical, these two nodes represent *ITA* identifiers.

Example. We take variables dfs in Fig. 1 as an example of *ITA*. First, the AST of this program is parsed. dfs at line 145 and dfs at line 146 are in the same scope while they are initialized variables with the same name. So, type inference tool is used to infer the types of these two variables. For dfs at line 145, the type is *List* which is inconsistent with the type of dfs at line 146. Thus, these two variables are *ITA* variables.

2.2.2. *Identification of IVT*

We use a type dependency graph [11] to help identify *IVT*. Dependency relation between attributes can be presented in the form of a dependency graph. Vertices represent attributes, and edges represent relations. The dependency graph is widely used in many research [11, 12]. We use a type dependency graph to describe the dependency relations between variable types. The type dependency graph in this paper is a directed graph $GT = (VT, ET)$. The vertices VT represent the set of specified variables with types $t \in T$ while the edges $ET \subseteq VT * VT$ represent static type dependency relations between variables in a program. Variables contain additional information, i.e. variable types and line numbers, modeled as vertex labels, while heuristic policies further confine relations. In the following, we list these heuristic policies:

Initiation policy: Given a statement s, like $x = c_1 OP_1 c_2 OP_2 \ldots c_i \ldots OP_k c_n$, $i \in [1, n]$. x is defined in this statement. OP_i represents operator in Python, e.g.

'+','−'. c_i is the smallest unit of an operand, e.g. a number, a string, or a function call. Then, $\forall i$, $i \in [1, n]$, type (x) is depended on type (c_i).

Define-use policy: Given two variables x_1, x_2, there exists a define-use relation between x_1 and x_2. x_1 is the definition and x_2 is its use. Then, type (x_2) depends on type (x_1).

Call policy: Given a function f, the type of the function call point is dependent on the function returned value.

According to the above policies, we can construct a type dependency graph. The next step is to evaluate the type dependency graph and decide *IVT*. In the type dependency graph, some vertices, e.g. some initial variable, are with known types while some vertices, e.g. some intermediate variables, are with unknown types. This can be obtained from the type inference tool. Types are propagated through edges. So, we take the depth-first strategy and reversely traverse all paths in this graph to find the vertices vt with the known types. Only the first vertex vt is recorded for each path. We mark them as vt_i, $i \in [1, n]$. Then, the decision condition can be expressed like the following equation:

$$\frac{|vt_1 \cap vt_2 \cdots \cap vt_i \cdots \cap vt_n| > 1}{IVT(id)}. \tag{1}$$

We can decide whether a variable is an *IVT* identifier by computing the above decision condition. If the result of a condition is *True*, this variable is an *IVT* identifier. Otherwise, this variable is not an *IVT* identifier.

Example. We take *hashes* at line 2458 in Fig. 2 as an example to explain the identification of *IVT*. Figure 3 provides the corresponding type dependency graph.

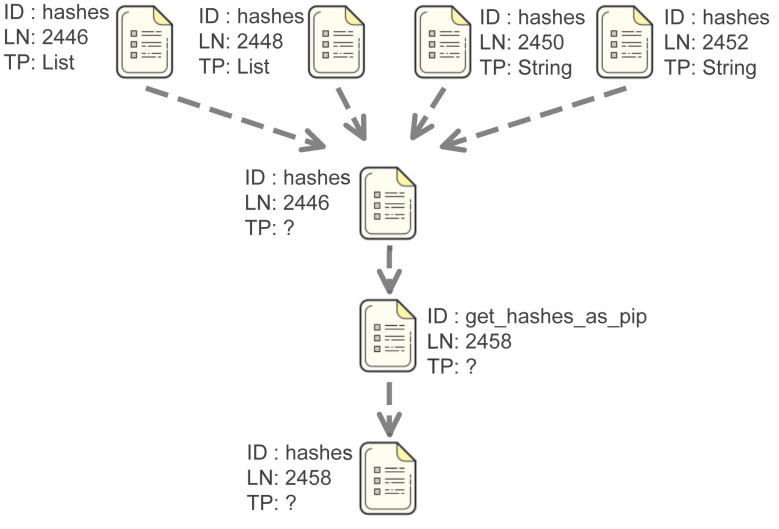

Fig. 3. Type dependency graph.

For each node in this graph, "ID" denotes identifier names. "LN" denotes line number. "TP" denotes type. Types transfer along with the arrow. If the types are unknown, we mark it as '?'. The variable *hashes* (line 2458) is initiated by the returned value of methods *get_hashes_as_pip*(). In the method *get_hashes_as_pip*(), the returned type of *hashes* (line 2453) depends on conditional structures. There are four potential type dependencies between returned value *hashes* (line 2453) and variables *hashes* (line 2446, line 2448, line 2450, line 2452). At line 2446 and line 2448, the types of variable *hashes* are initiated as *List*. At line 2450 and line 2452, the types of variable *hashes* are initiated as *String*. After obtaining this type dependency graph, we reversely traverse all nodes from node *hashes* at line 2458. We find all nodes with known types in four paths. Then, the decision condition 1 is computed, and the result is *True*. Therefore, the variable *hashes* (line 2458) is an *IVT* identifier.

3. Study Design

This section introduces the whole study design of our paper, including dataset collection, tool implementation, research questions, and the corresponding analysis methods.

Figure 4 presents the overview of this study. We first use our proposed approach to identify dynamic typing in Python projects on the given dataset. Then, we compute the frequency and distributions of the usage of dynamic typing. Next, we separately investigate the relations between dynamic typing and change-proneness, maintenance time, size of commit files, and the acceptance of commits.

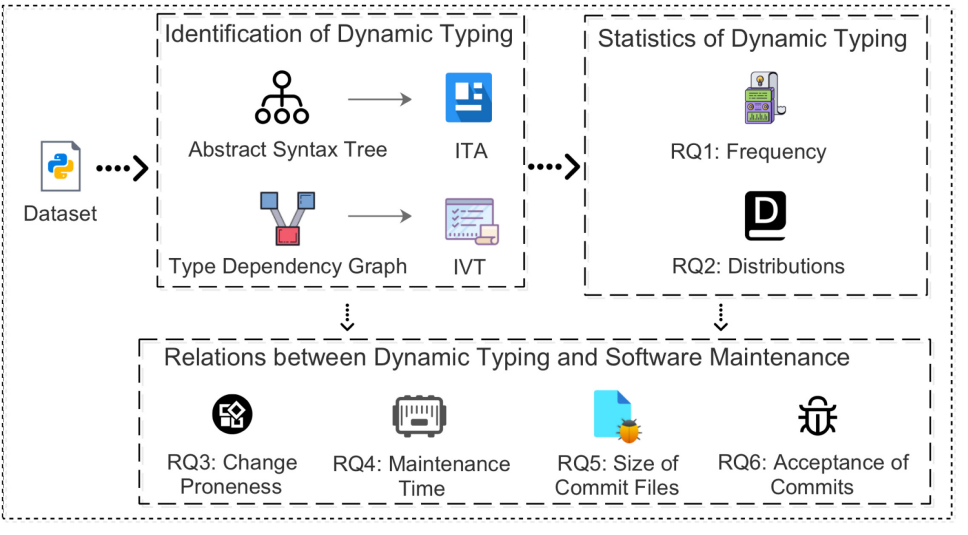

Fig. 4. Overview of study design.

3.1. *Data collection*

We collected the dataset from GitHub.[c] We first sorted all Python projects on GitHub according to their stars. Then, we randomly selected one of the most representative open-source projects in 10 different areas, such as data analysis, machine learning, web applications, from the top-50 projects of the most stars as our datasets. All source codes and related pulls and commits are included. The selected 10 projects are as follows:

- Ansible: Ansible is a simple IT automation system. It can handle configuration management, application deployment, cloud provisioning, and so forth.
- Cookiecutter: Cookiecutter provides Python project templates. Developers can create projects from these project templates like Python package projects.
- Dash: Dash is usually used to directly tie modern UI elements like dropdowns, sliders, and graphs to analytical Python code.
- FaceSwap: FaceSwap is an application designed to utilize deep learning to recognize and swap faces in pictures and videos.
- mitmproxy: mitmproxy is interactive, SSL/TLS-capable intercepting proxy with a console interface to deal with HTTP/1, HTTP/2, and WebSockets.
- TensorFlow-Models: TensorFlow is a deep learning framework developed by Google Inc. The TensorFlow-Model contains the implementation of many widely-used models.
- pandas: pandas provides fast, flexible, and expressive data structures to make working with "relational" or "labeled" data both easy and intuitive.
- Pipenv: Pipenv is a tool to automatically create and manage a virtual environment for projects and adds and removes packages from Pipfile when installing or uninstalling packages.
- Requests: Requests is an HTTP library for Python. Programmers can send HTTP/1.1 requests easily through Requests.
- Sentry: Sentry is a service that helps monitor and fix crashes in real-time.

Table 1 provides details of these 10 selected projects. TensorFlow-Models holds the most stars (65.2 k) among 10 projects, while Ansible holds the most commits (44,908) and pull requests (42,724). Sentry owns the most files (2806) in these projects. Pandas have the most lines of code (LoC) (305) for each file.

3.2. *Tool implementation*

We implement our proposed approaches in Sec. 2 with Python 3.9 into a tool to identify *ITA* and *IVT*. Details of the implementation are described as follows:

(1) For *ITA*, we traverse ASTs of the programs in a depth-first order and use type inference tool *PySonar*2 [13] to obtain variables and types. *PySonar*2 is a state-of-the-art type inference tool that performs whole-project inter-procedural analysis to

[c]https://github.com.

Table 1. The summary of dataset.

Projects	Stars	Commits	Pulls	Files	AVG LoC
Ansible	44.1 k	44,908	42,724	1428	149
Cookiecutter	12.5 k	2150	840	83	69
Dash	12.5 k	2750	438	86	151
FaceSwap	31.5 k	956	417	183	199
mitmproxy	19.5 k	6920	1700	531	97
TensorFlow-Models	65.2 k	3975	3031	1766	182
pandas	25.9 k	20,436	15,952	1107	305
Pipenv	20.7 k	5343	1240	999	241
Requests	43.1 k	4469	2010	35	198
Sentry	25.6 k	28,807	15,059	2806	91

infer types of variables. *PySonar*2 is widely used in industry, i.e. Google, Sourcegraph, and Academia [14] as the foundation of work related to type analysis [9]. To identify *ITA* identifiers, we first use Python *ast* module to parse the given program into an AST. Then, the AST is divided into subtrees based on different scopes. We traverse the ASTs and record all information of variable nodes, including names, line numbers, and so forth. Next, we use *PySonar*2 to infer the type information of these variables. Finally, we locate all identifiers with the same name and compare their types. If the types are inconsistent, these identifiers are determined to be *ITA* identifiers.

(2) For *IVT*, we construct a type dependency graph and evaluate the graph to return *IVT* identifiers. To construct the type dependency graph, the AST is also traversed in a depth-first order to obtain type dependency relations of initiation policy and call policy. We locate all assignment statements and function calls. We set up a dependency relation between the variables performing *read* operation and variables performing *write* operation for each assignment statement. We also set up a dependency relation between returned values and their call points for each function call. Then, we use *beniget* module [15] of Python to obtain type dependency relations of define-use policy. After that, we view these type dependency relations as the edges, view the variables as the vertices, and then, set up a type dependency graph. In the type dependency graph, we compute the decision condition 1 to return *IVT* identifiers.

The whole experiment is conducted on a computer of Ubuntu 18.10 (×86_64) with Inter (R) Core (TM) i7-6500U CPU @ 2.50 GHz.

3.3. *Research questions and analysis methods*

RQ1: *How often is dynamic typing applied in real programming projects of Python?*

This research begins with the frequencies of dynamic typing identifiers in programming practice and type distributions in all types of dynamic typing identifiers. A quantitative observation on frequencies of dynamic typing is presented in this research question.

To complete this research question, we classify all types of identifiers into 11 categories based on the standard Python typing system [16], including *List*, *Int*, *Object*, *String*, *Callable*, *Bool*, *Float*, *Dictionary*, *Set*, *Tuple*, and *None*. We merge all *Object* types, e.g. user-defined class, into object categories and all function returned values into callable categories. Types with minor numbers, e.g. complex numbers, are ignored. We scan all Python files in the dataset and discover all *ITA* and *IVT* identifiers with the implemented tool. Then, we provide an overall distribution of dynamic typing identifiers, including the number of *ITA* and *IVT* in different measurements (e.g. LoC, per file).

RQ2: *How does dynamic typing distribute in different Python program structures?*

This research question investigates the distributions of dynamic typing in different program structures. Here, we divide six structures into four categories according to the grammar specification [17], including conditional structures (If), loop structures (For, While), exception structures (Try), module structures (Function, Class). Numbers are counted for dynamic typing identifiers related to these structures.

To identify the above program structures, we first parse all programs to obtain their ASTs. Then, we locate six structures in ASTs. For instance, the scopes of these structures are recorded from line 3 to line 5. Finally, we decide whether *ITA* and *IVT* are contained in these structures and give the statistic distributions of dynamic typing identifiers on these structures.

RQ3: *What is the relation between dynamic typing and change-proneness?*

This research question concentrates on the relations between dynamic typing and change-proneness. Change-proneness refers to whether a Python file is modified during whole software maintenance. If a user modifies a file and changes its contents in the GitHub system, they submit a commit to GitHub. The null hypothesis is H03: *The more a file contains dynamic typing identifiers, the greater the possibility is for the file to be changed.*

In this research question, *Spearman's rank correlation coefficient* [18] is computed to measure the relation. *Spearman's rank correlation coefficient* is a non-parametric measure of rank correlation in statistics, which assesses how well the relations between Variable X and Variable Y can be described. If the value of *Spearman's rank correlation coefficient* is between 0 and 1, it is a positive monotonic correlation between X and Y. If the value is between -1 and 0, it is a negative monotonic correlation between X and Y. The bigger the absolute value of *Spearman's rank correlation coefficient* is, the stronger the relationship is. 0 represents no monotonic correlation between X and Y.

We collect all commit files of 10 projects and record them. Non-Python files are excluded. For simplicity, we use the newest version of projects as our baselines. The number of *ITA* and *IVT* identifiers of each file in a project can be seen as Variable X of *Spearman's rank correlation coefficient*. The times of the file appearing in all commit files can represent change-proneness, which can be seen as Variable Y of *Spearman's rank correlation coefficient*. *Spearman's rank correlation coefficient* is then computed.

RQ4: *What is the relation between dynamic typing and maintenance time?*

This research question aims to investigate whether dynamic typing has an impact on maintenance time. The null hypothesis is tested: H04: *The more dynamic typing identifiers participate in software maintenance, the longer the maintenance time is.*

Spearman's rank correlation coefficient is also used to measure this relation. Issues usually include the whole software maintenance procedure on GitHub, e.g. the descriptions of exceptional behavior, fixing solutions, and pull requests. When an exceptional behavior is reported in an issue, developers discuss and determine whether the exceptional behavior is a real bug. Once the bug is confirmed, developers modify source code in local repositories. Before their modified code is merged into a branch, a pull request can be open to tell others about the changes they have pushed to the branch. Other developers can discuss and review the potential changes with collaborators and add follow-up commits. After that, problems are fixed, and issues are closed. The software maintenance procedure is over.

Pull requests usually have connections with one or more issues and one or more commits. We exclude some pull requests that have no connections with any issue. In this research question, we collect all pull requests from GitHub by a crawler. Then, we match the numbers of related issues by regular expression. According to these numbers of issues, we collect each issue's start time and end time. Maintenance time is computed by the difference between the first issue's start time and the last issue's end time. We exclude all pull requests with maintenance time over a year. Maintenance time can be seen as Variable Y of *Spearman's rank correlation coefficient*. For each pull request, we collect related commits by GitHub API. We find all *ITA* and *IVT* identifiers in these commit files, which can be seen as Variable X of *Spearman's rank correlation coefficient*.

RQ5: *What is the relation between dynamic typing and the size of commit files?*

In this research question, we investigate whether dynamic typing identifiers have relations with the size of commit files during software maintenance. We test the null hypothesis: H05: *The more dynamic typing participates in commit files, the larger the size of commit files is.*

We use the number of LoCs of commit files, excluding blank lines, to represent the size of commit file, which can be seen as Variable Y of *Spearman's rank correlation coefficient*. The number of *ITA* and *IVT* identifiers in these commit files can be seen as Variable X. Then, we can compute *Spearman's rank correlation coefficient* between X and Y.

RQ6: *What is the relation between dynamic typing and the acceptance of commits?*

This research question focuses on the relations between dynamic typing and the acceptance of commits. We investigate this research question by testing the null hypothesis: H06: *Dynamic typing in commit files helps improve the acceptance of this commit during software maintenance.*

In this research question, we utilize *Fisher's exact test* [19]. *Odds ratios* (*ORs*) is computed to indicate whether there are relations between the acceptance of commits and dynamic typing. Acceptance of commits refers to whether the projects successfully accept a commit. When a commit is reviewed, a state is returned. This state could be *success*, *error*, *pending*, *failure*.

An *OR* is a statistic that quantifies the strength of the association between two events X, e.g. dynamic typing, and event Y, e.g. the acceptance of commits. The *OR* is defined as the ratio of the odds m of an event occurring in one sample, e.g. a commit with returned state of *success* to the odds n of it occurring in the other sample, e.g. the related commit files containing *ITA* and *IVT* identifiers. The computation of *OR* is as follows:

$$OR = \frac{m/(1-m)}{n/(1-n)}. \tag{2}$$

X and Y are independent if and only if the *OR* equals 1. If the *OR* is greater than 1, X and Y are correlated. Conversely, if the *OR* is less than 1, X and Y are negatively correlated. For commits, whether they are in a state of success and whether they contain *ITA* and *IVT* can be taken as input to the computation of *OR*. Then, we can depict the relationship between the acceptance of commits and dynamic typing by *OR*.

4. Findings

This section presents the answer to all research questions, lists our study's results, and gives eight findings from these results.

4.1. *Answering RQ1: Frequency of dynamic typing identifiers in Python*

Table 2 shows the statistics of *ITA* and *IVT* identifiers. For each row, we list the total number of dynamic typing identifiers (Total), the average number of LoC containing a dynamic typing identifier, the number of files within at least one dynamic typing identifier (Num), and the percentage in all files (pct.). The average numbers of these items (AVG) are computed in the last row.

As shown in Table 2, project *Pipenv* owns the most usages of dynamic typing, including 3074 *ITA* and 2530 *IVT* identifiers. On average, there are 861.4 *ITA* for each project. There is one *ITA* identifier for every 172 LoC. In a project, 131.4

Table 2. Frequency of dynamic typing identifiers in projects.

	ITA		IVT		Files with ITA		Files with IVT	
Projects	Total	LoC/ITA	Total	LoC/IVT	Num	pct.	Num	pct.
Ansible	1472	144.5	2026	105.0	233	16.3%	176	12.3%
Cookiecutter	22	260.3	25	229.1	7	8.4%	7	8.4%
Dash	42	309.2	25	519.4	9	10.5%	3	3.5%
FaceSwap	195	186.8	94	387.4	40	21.9%	18	9.8%
mitmproxy	264	195.1	27	1907.7	55	10.4%	7	1.3%
TensorFlow-Models	1333	241.1	898	357.9	278	15.7%	166	9.4%
pandas	1555	217.1	28	12,058.4	201	18.2%	8	0.7%
Pipenv	**3074**	78.3	**2530**	95.2	320	32.0%	220	22.0%
Requests	61	113.6	81	85.6	10	28.6%	7	20.0%
Sentry	596	428.4	656	389.2	161	5.7%	104	3.7%
AVG	**861.4**	**172.0**	**639.0**	**231.8**	**131.4**	**14.6%**	**71.6**	**7.9%**

Python files averagely contain *ITA* identifiers accounting for 14.6% of all Python files.

Finding 1: 14.6 % Python files contain ITA identifiers . On average, there is one ITA for every 172.0 LoC.

There are 639.0 *IVT* identifiers for each project on average. There is one *IVT* identifier for every 231.8 LoC. In a project, 71.6 Python files averagely contain *IVT* identifiers accounting for 7.9% of all Python files.

Finding 2: 7.9 % Python files contain IVT identifiers. On average, there is one IVT for every 231.8 LoC.

4.2. *Answering RQ2: Distributions of dynamic typing identifiers in Python*

In Table 3, four category structures of Python programs are listed at the first row, including condition structures (i.e. *If*), loop structures (i.e. *For*, *While*), exception structures (i.e. *Try*), and module structures (i.e. *Function*, *Class*). The number in this table represents the number of dynamic typing identifiers related to these structures.

Results show that 479.4 dynamic typing identifiers, on average, are related to conditional structure *If*, which is around five times of loop structure *For* and *While* is eight times of exception handling structure *Try*. On average, 656.5 dynamic typing identifiers are in function structures, twice that in the class structure.

Finding 3: The number of dynamic typing identifiers is more in condition structures than in loop structures and exception handling structures. Dynamic typing is more likely to appear in function structure than class structure.

Table 3. Relations between structures and dynamic typing identifiers in projects.

	Cond.	Loop		Excep.	Mod.	
Projects	If	For	While	Try	Func.	Class
Ansible	1008	203	16	212	1329	727
Cookiecutter	19	6	1	7	21	0
Dash	11	4	1	1	21	10
FaceSwap	119	22	5	6	181	153
mitmproxy	113	17	11	23	179	106
TensorFlow-Models	705	214	27	4	1052	333
pandas	923	104	12	70	1310	865
Pipenv	1503	184	90	243	1904	1217
Requests	41	4	2	11	49	32
Sentry	352	78	4	73	519	317
AVG	**479.4**	83.6	16.9	65	**656.5**	376

4.3. *Answering RQ3: Relations between dynamic typing identifiers and change-proneness*

Table 4 summarizes the results of *Spearman's rank correlation coefficient* between change-proneness and *ITA*, *IVT*. For each row, we list *Spearman's rank correlation coefficient* and *p-value* on each project. We highlight all rows that *p-value*s are less than 0.05 in bold.

Table 4. Relations between change-proneness and dynamic typing identifiers in projects.

	ITA		IVT	
Projects	Spearman	*p*-value	Spearman	*p*-value
Ansible	−0.007	9.16E−01	0.087	2.49E−01
Cookiecutter	−0.177	7.04E−01	−0.018	9.69E−01
Dash	0.500	1.70E−01	0.500	6.67E−01
FaceSwap	0.149	3.60E−01	0.191	4.49E−01
mitmproxy	0.086	5.31E−01	−0.296	5.18E−01
TensorFlow-Models	0.053	3.81E−01	0.100	1.98E−01
pandas	**0.488**	**1.91E−13**	0.345	4.03E−01
Pipenv	0.021	7.04E−01	−0.030	6.61E−02
Requests	**0.644**	**4.44E−02**	0.393	3.83E−01
Sentry	**0.192**	**1.46E−02**	0.133	1.77E−01

For the results of *ITA* and change-proneness, the *p-value*s of three projects are less than 0.05, while the *Spearman's rank correlation coefficient*s are between 0 and 1. However, for the results of *IVT* and change-proneness, *p-value*s of all projects are greater than 0.05. The results are not significant. Hence, we conclude that for fewer projects, there are positive correlations between change-proneness and *ITA*. Meanwhile, in most cases, there is no relation between dynamic typing and change-proneness.

Finding 4: There is a weak relation between dynamic typing and change-proneness in most cases.

4.4. *Answering RQ4: Relations between dynamic typing identifiers and maintenance time*

Table 5 reports the results between dynamic typing and maintenance time. For each row, we list *Spearman's rank correlation coefficient* and *p-value* on each project. We highlight all cells that *p-value*s are less than 0.05 in bold.

From Table 5, for the results of *ITA* and maintenance time, *p-value*s of six projects are less than 0.05 while the *Spearman's rank correlation coefficient*s of them are between −1 and 0. For the results of *IVT* and maintenance time, *p-value*s of the same six projects are less than 0.05 while the *Spearman's rank correlation coefficient*s of them are also between −1 and 0. We can conclude that there are probably negative

Table 5. Relations between maintenance time and dynamic typing identifiers in projects.

	ITA		IVT	
Projects	spearman	*p*-value	spearman	*p*-value
Ansible	**−0.610**	**1.88E−280**	**−0.621**	**1.53E−274**
Cookiecutter	**−0.362**	**1.62E−05**	**−0.401**	**1.47E−06**
Dash	0.035	7.24E−01	−0.153	1.44E−01
FaceSwap	−0.275	2.69E−01	−0.274	4.76E−01
mitmproxy	**−0.214**	**1.22E−02**	**−0.689**	**1.71E−07**
TensorFlow-Models	**−0.537**	**3.38E−06**	**−0.590**	**1.07E−06**
pandas	**−0.138**	**8.88E−27**	**−0.400**	**1.86E−71**
Pipenv	0.101	6.16E−02	−0.081	1.91E−01
Requests	0.037	4.71E−01	0.044	3.96E−01
Sentry	**−0.298**	**5.29E−07**	**−0.418**	**9.89E−11**

correlations between dynamic typing and maintenance time. There is a high probability that maintenance time decreases as dynamic typing identifiers increase in projects.

Finding 5: There is a high probability that maintenance time decreases as dynamic typing identifiers increase in projects.

4.5. *Answering RQ5: Relations between dynamic typing identifiers and the size of commit files*

Table 6 reports the relations for dynamic typing and size of commit files. Similarly, for each row, we list *Spearman's rank correlation coefficient* and *p-value* on each project. We highlight all cells that *p-value*s are less than 0.05 in bold.

From Table 6, we can notice that three projects are not significant. Their *p-value*s are greater than 0.05. For the rest seven significant projects, *Spearman's rank correlation coefficient*s of four projects are between 0 and 1, while *Spearman's rank correlation coefficient*s of three projects are between −1 and 0. Meanwhile, the absolute values of *Spearman's rank correlation coefficient*s of these seven projects

Table 6. Relations between size of commit files and dynamic typing identifiers.

	ITA		IVT	
Projects	spearman	*p*-value	spearman	*p*-value
Ansible	**0.020**	**1.26E−02**	0.013	1.67E−01
Cookiecutter	−0.055	3.37E−01	−0.013	8.26E−01
Dash	*−0.148*	**1.54E−02**	−0.004	9.58E−01
FaceSwap	−0.073	7.07E−02	−0.042	5.37E−01
mitmproxy	−0.047	1.40E−01	−0.157	1.87E−01
TensorFlow-Models	**0.136**	**1.18E−13**	**0.089**	**8.54E−05**
pandas	**0.031**	**3.46E−06**	0.090	9.63E−02
Pipenv	*−0.065*	**1.21E−03**	*−0.085*	**5.11E−04**
Requests	*−0.117*	**9.49E−07**	**0.055**	**3.57E−02**
Sentry	**0.120**	**3.14E−22**	0.015	3.11E−01

are all less than 0.2 and close to 0. For the results of *IVT* and size of commit files, *p-value* of three projects are less than 0.05. The *Spearman's rank correlation coefficient*s of two projects among them are between 0 and 1. Similar to the results of *ITA* and size of commit files, absolute values of *Spearman's rank correlation coefficient*s of all these three projects are less than 0.2 and close to 0. We can conclude that the correlation is low between dynamic typing and the size of commit files in most cases.

Finding 6: Most cases show little connection between dynamic typing and the size of commit files.

4.6. *Answering RQ6: Relations between dynamic typing identifiers and the acceptance of commits*

Table 7 summarizes Fisher's exact test results and *OR*s when testing H. Each row represents *OR* value, *p-value* on each project. In the table, we highlight all *p-value*s less than 0.05 in bold, *OR* less than 1 in bold and *OR* greater than 1 in italic.

Table 7. Relations between the acceptance of commits and dynamic typing identifiers.

	ITA		IVT	
Projects	OR	*p*-value	OR	*p*-value
Ansible	**0.632**	**1.14E−29**	**0.632**	**2.16E−26**
Cookiecutter	**0.486**	**3.72E−03**	**0.486**	**3.72E−03**
Dash	**0.725**	**4.64E−02**	0.769	1.16E−01
FaceSwap	**0.469**	**3.46E−02**	**0.305**	**4.07E−04**
mitmproxy	**0.764**	**1.08E−02**	0.791	4.52E−01
TensorFlow-Models	**0.758**	**1.50E−03**	0.937	5.18E−01
pandas	*1.064*	**1.48E−02**	*1.272*	**7.68E−06**
Pipenv	0.847	5.90E−02	0.809	2.35E−02
Requests	*1.472*	**1.49E−03**	*1.430*	**2.33E−03**
Sentry	**0.905**	**1.09E−02**	**0.926**	**6.84E−02**

*OR*s vary across projects, within each project, across different dynamic typing practices. For the results of *ITA* and the acceptance of commits, nine projects are significant in which *p-value*s are less than 0.05. Among these nine projects, *OR* values of seven projects are less than one while *OR* values of two projects are greater than 1. For the results of *IVT* and the acceptance of commits, six projects are significant in which *p-value*s are less than 0.05. Among these six projects, *OR* values of four projects are less than one while *OR* values of two projects are greater than 1. Thus, we conclude that, in most cases, *ITA* and *IVT* harm the acceptance of commits.

Finding 7: There is a high probability that dynamic typing decreases the acceptance of commits.

5. Discussions

According to the findings in Sec. 4, we further summarize the broader implications and list the threats to validity for our research.

5.1. *Implications*

We discuss implications based on our results, findings, and observations. In particular, we draw lessons from the findings for the developers and future researchers.

5.1.1. *Implications for developers*

Developers have a widespread debate over the choice between dynamic typing and static typing [3, 9]. This also provides related implications and new ideas about this topic. Our empirical experiments provide statistical evidence to demonstrate that **dynamic typing is actually beneficial but dangerous to software maintenance**. Findings 1 and 2 reveal that dynamic typing, i.e. *IVT* and *ITA*, is often used in Python software development and maintenance. The usages of *ITA* identifiers are more frequent than *IVT* identifiers in programming practices. On the one hand, Finding 5 suggests a high probability that maintenance time decreases as dynamic typing identifiers increase in projects, which implies that dynamic typing identifiers may benefit software maintenance. The usage of dynamic typing identifiers may help improve the speed of writing patches by simplifying and shortening the source code to save maintenance time [20]. On the other hand, from Finding 7, dynamic typing decreases the acceptance of commits, making them dangerous. This could be caused by the communication errors between dynamic typing identifiers with other existing identifiers in new added patches [9].

5.1.2. *Implications for researchers*

Due to the lack of type declarations in dynamic languages, type issues, e.g. type inference [14, 21], type system [22, 23], are widely concerned in the research area of programming languages. Our experiment results also implicate potential directions for future research. Due to code readability and program logic, programmers still have habits using most variables as static ones. (Finding 1 and Finding 2) **For a programming language, it may be good to separate dynamic variables and static variables according to the different needs of users.** These two kinds of identifiers can be explicitly denoted. The static identifiers can improve readability and efficiency [24], while the dynamic identifiers can make the language more flexible.

Weak relations between dynamic typing and change-proneness (Finding 4) imply that most dynamic typing identifiers maybe not the direct root causes of software bugs during software developments. In RQ2, we find that dynamic typing may relate to certain structures. (Finding 3) One possible explanation is that dynamic typing-related bugs may only be triggered by a few certain usage patterns of dynamic typing identifiers, e.g. dynamic typing related to certain structures. For example, CPython

bug 42763[d] exposes an error of the CPython interpreter. In function *threading. Thread(target=do_work)*, parameter *target* adopts an argument *do_work*. While the type of *do_work* is *int* rather than *function*, the program will trigger an error of the CPython interpreter. Similar bug-triggering patterns may help experienced developers quickly identify the root causes of software bugs, reducing software maintenance time. Therefore, **it may be a good research direction to mining the patterns of dynamic-typing bugs and their fixes for future researchers**. The summary of these patterns may assist developers in achieving higher efficiency in software developments and maintenance.

5.2. *Threats to validity*

The work presented in this paper was carefully planned and executed, but there exist several threats to validity. This section discusses them and the mitigation strategies.

5.2.1. *Threats to internal validity*

Threats to *internal validity* concern confounding factors that could influence our results. Our research relies on inferred results of *PySonar2*. Although we considered the situation that the inferred results are "?" in our tool, inaccurate inferred results from *PySonar2* still exist. Besides, a few files that *PySonar*2 cannot analyze due to the failure of AST parse are excluded. We assume that all these files have little impact on our results. To reduce the potential bias, we pick popular and representative projects on GitHub as our dataset and try our best to improve the approaches on identifying dynamic typing to lower the influence of our results.

5.2.2. *Threats to external validity*

Threats to *external validity* concern the generalizability of our results. Our dataset is collected from GitHub. Different projects may own different naming and programming habits. It is difficult to use 10 projects to represent all projects in whole Python programming practices. It is probably a potential threat. To mitigate this threat, we select the most representative projects on different areas from more than 10 k stars open-source Python projects. We are confident that the results from selected projects are reliable.

5.2.3. *Threats to construct validity*

Threats to *construct validity* concern how we set up our study. This paper set up six research questions to measure the relations between dynamic typing and software maintenance. These research questions are answered by quantitative analysis. A single project may bring potential deviation to our result. To mitigate potential

[d]https://bugs.python.org/issue42763.

issues, we use Spearman's rank correlation coefficients and Fisher's exact test to analyze 10 big projects with different topics. We believe the results of our study are reliable.

6. Related Work

Prior work on the impacts of programming language features on project maintenance falls into the following four categories:

6.1. *Programming habits*

Many researches have been done on good programming habits [25–27], e.g. programming idioms [28–30] and bad programming habits [31], e.g. code smells [32, 33].

Alexandru *et al.* [34] built a catalog of *pythonic idioms* and researched the features of code in Python. They classified the idioms in two characteristics: readability and performance, and frequency were computed in 1000 open-source projects. These idioms in Python code make Python code more readable and flexible. Rodrigues Jr. and Terra [35] investigated how developers use dynamic features based on 28 open-source Ruby projects. Results show that dynamic feature usage in Ruby ranges from 2.08% to 3.08% at a 95% confidence interval. In the work of Alexandru and Rodrigues Jr., they found and classified code patterns with dynamic features. Although the frequency is not high, dynamic features make programming flexible in the dynamic programming language, bringing potential threats to software maintenance. Hanenberg *et al.* [3] conducted an empirical study on the impact of static typing on software maintenance. They investigated whether static type systems improve the maintainability of software systems in terms of understanding undocumented code, fixing type errors, and fixing semantic errors. The results showed that static types are beneficial to these activities, except for fixing semantic errors.

In our research of this paper, we investigate dynamic typing and software maintenance to figure out whether dynamic typing leads to bad results on software maintenance. Besides, we also analyze relations between dynamic typing and software maintenance from multiple angles, e.g. change-proneness, maintenance time.

6.2. *Change-proneness*

Change-proneness is also an essential measurement for software quality.

Kyriakakis *et al.* [36] investigated the frequency of possible dynamically extendible code patterns (e.g. through method invocation) in 10 milestone PHP projects to figure out change-proneness of dynamic feature pattern instances in PHP applications. Their results showed that methods employing irregular patterns are less change-prone. Chen *et al.* [37] conducted a comprehensive study to explore and validate the characteristics of feature changes in Python. They investigated change occurrences in 85 projects. They found that dynamic features, e.g. dynamic

attributes, are increasingly used, making the code changeable, and dynamic feature code plays both positive and negative roles in maintenance activities. Wang *et al.* [38] made an empirical study on the impact of Python dynamic features, e.g. dynamic attributes *setattr*, *del*, on change-proneness. Historical data from 4 to 7 years of the development of seven open-source systems was analyzed. The results showed that files with these dynamic features are more change-prone. In the work of Chen and Wang, dynamic attributes are the main research objects. Other dynamic features, e.g. dynamic typing, are not involved. In contrast, our study presents the relations between dynamic typing and change-proneness by mining the commits on 10 open-source projects.

6.3. *Maintenance time*

Maintenance time is closed to the efficiency of software development. Prechelt [39] comparatively analyzed 80 implementations of programs in seven different languages (C, C++, Java, Perl, Python, Rexx, and Tcl). Their research showed that programs implemented in dynamic programming languages (Perl, Python, Rexx, and Tcl) took half or less time to write than equivalent programs in static programming languages (C, C++, Java). However, this study did not attempt to measure the maintenance time of these programming languages. The maintenance time is also an essential aspect of programming language effectiveness. Denny *et al.* [40] investigated common syntax errors in programming practices. They classified the amount of time spent fixing a variety of syntax errors in the computer programming course in Java. Their results showed that type mismatch errors, e.g. trying to assign a double value to an integer, accounted for approximately 18.4% of the time novices spend identifying and fixing compiler errors, second only to "cannot resolve identifier" errors, in the amount of time taken. Their experiment was conducted on a static typing system of Java. Different from the above works, we analyze the relations between dynamic typing and maintenance time in Python. The result shows that the usage of dynamic typing may reduce maintenance time, improving development efficiency.

6.4. *Software bugs*

Researches on software bugs concern bug classification [41], bug prediction [42], bug detection [43, 44], and bug fixing [45, 46].

Ray *et al.* [47] conducted a large-scale study by combining multiple regression modeling with visualization and text analytics to study the effect of language features on software quality. They found that programs in static programming language have better code quality than in dynamic programming language. Chen *et al.* [48] made a study on the changes of dynamic features of code when fixing bugs. They analyzed the changes of dynamic feature code and the roles of dynamic features in 17,926 bug-fix commit in 17 Python projects. Their research showed that the changes of dynamic feature code are significantly related to bug-fix activities rather than non-

bug-fix activities. In the work of Ray and Chen, a fact is revealed that dynamic features may bring potential threats to software quality. In this paper, we investigate whether dynamic typing has an impact on software quality.

Chen *et al.* [9] conducted an empirical study on nine open-source Python projects to understand dynamic typing-related practices. They investigate whether their usage correlates with the increased likelihood of bugs occurring and how developers fix dynamic typing-related bugs. The results show that dynamic typing has a significant positive correlation with bug occurring, and type checks or exception handling are usually added to fix dynamic typing related bugs. Their work has demonstrated the correlation between dynamic typing and bug occurrence, while our work extends their research by mining software repositories to quantify the correlation between dynamic typing and software maintenance. We investigate the relation between dynamic typing and change-proneness, maintenance time, size of commit files, and the acceptance of commits.

7. Conclusion

Python is well known as a typical dynamic language with dynamic features. The usage of dynamic typing may affect software maintenance. This paper performs an empirical study to investigate how dynamic typing impacts software maintenance. We propose the AST analysis and the type dependency graph to identify two dynamic typing: ITA and IVT. We summarize six research questions to investigate the relations between dynamic typing and software maintenance tasks and collect a dataset containing the 10 most popular projects on GitHub for empirical experiments. The results show a weak connection between change-proneness and variable dynamic typing. But there is a high probability that maintenance time and the acceptance of commits decreases as dynamic typing identifiers increase in projects, which implies that dynamic typing is actually beneficial but dangerous to software maintenance.

This research also provides implications for software developers and researchers. Researchers may divide dynamic variables and static variables for research on programming languages while developing new programming languages to fit programming habits and improve program efficiency. Dynamic typing identifiers may not be the direct root causes for most software bugs. The categories of these bugs are worth exploring. In the future, we also will conduct more profound research on the features of dynamic typing-related bugs. Our tool, experimental data, and results are publicly available at https://github.com/xiaxinmeng/DynamicTypinginPython.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This research was supported by NSFC 61832009, the program B for Outstanding Ph.D. candidate of the Nanjing University.

References

1. L. Tratt, Dynamically typed languages, *Adv. Comput.* **77** (2009) 149–184.
2. G. Richards, S. Lebresne, B. Burg and J. Vitek, An analysis of the dynamic behavior of JavaScript programs, *ACM SIGPLAN Not.* **45** (2010) 1–12.
3. S. Hanenberg, S. Kleinschmager, R. Robbes, E. Tanter and A. Stefik, An empirical study on the impact of static typing on software maintainability, *Empir. Softw. Eng.* **19**(5) (2014) 1335–1382.
4. Pytorch, https://github.com/pytorch/pytorch.
5. Django, https://github.com/django/django.
6. Ansible, https://github.com/ansible/ansible.
7. Ieee rank, https://spectrum.ieee.org/top-programming-languages/.
8. A. Holkner and J. Harland, Evaluating the dynamic behaviour of Python applications, in *Proc. Thirty-Second Australasian Conf. Computer Science* Vol. 91, 2009, pp. 19–28.
9. Z. Chen, Y. Li, B. Chen, W. Ma, L. Chen and B. Xu, An empirical study on dynamic typing related practices in Python systems, in *Proc. 28th Int. Conf. Program Comprehension*, 2020, pp. 83–93.
10. M. Pradel, P. Schuh and K. Sen, TypeDevil: Dynamic type inconsistency analysis for JavaScript, *Int. Conf. Software Engineering*, 2015, pp. 314–324.
11. R. Hoover, Incremental graph evaluation, Technical report, Cornell University (1987).
12. M. He and J. Zhang, A dependency graph approach for fault detection and localization towards secure smart grid, *IEEE Trans. Smart Grid* **2**(2) (2011) 342–351.
13. PySonar2, https://github.com/yinwang0/pysonar2.
14. Z. Xu, X. Zhang, L. Chen, K. Pei and B. Xu, Python probabilistic type inference with natural language support, in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2016, pp. 607–618.
15. beniget, https://github.com/serge-sans-paille/beniget/.
16. Python type system, https://docs.python.org/3.9/reference/datamodel.html#objects-values-and-types.
17. Python grammar specification, https://docs.python.org/3.9/reference/grammar.html.
18. C. Spearman, The proof and measurement of association between two things, *Am. J. Psychol.* **15**(1) (1904) 72–101.
19. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures* (Chapman & Hall/CRC, Boca Raton, 2020).
20. L. D. Paulson, Developers shift to dynamic programming languages, *Computer* **40**(2) (2007) 12–15.
21. M. K. Azerounian, J. S. Foster and B. Min, Simtyper: Sound type inference for Ruby using type equality prediction, *Proc. ACM Program. Lang.* **5**(OOPSLA) (2021) 1–27.
22. J. P. Near *et al.*, Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy, *Proc. ACM Program. Lang.* **3**(OOPSLA) (2019) 1–30.
23. M. Toro, R. Garcia and É. Tanter, Type-driven gradual security with references, *ACM Trans. Program. Lang. Syst.* **40**(4) (2018) 1–55.
24. R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva, Ranking programming languages by energy efficiency, *Sci. Comput. Program.* **205** (2021) 102609.
25. A. Vizcaíno, J. Contreras, J. Favela and M. Prieto, An adaptive, collaborative environment to develop good habits in programming, in *Proc. Int. Conf. Intelligent Tutoring Systems*, 2000, pp. 262–271.
26. J. Spacco, D. Fossati, J. Stamper and K. Rivers, Towards improving programming habits to create better computer science course outcomes, in *Proc. ACM Conf. Innovation and Technology in Computer Science Education*, 2013, pp. 243–248.

27. O. Meerbaum-Salant, M. Armoni and M. Ben-Ari, Habits of programming in scratch, in *Proc. Annual Joint Conf. Innovation and Technology in Computer Science Education*, 2011, pp. 168–172.
28. D. Harel, G. Katz, A. Marron and G. Weiss, The effect of concurrent programming idioms on verification: A position paper, *Int. Conf. Model-Driven Engineering and Software Development*, 2015, pp. 363–369.
29. D. Harel, G. Katz, R. Lampert, A. Marron and G. Weiss, On the succinctness of idioms for concurrent programming, *Int. Conf. Concurrency Theory*, 2015, pp. 85–99.
30. J. Coplien, C++ idioms, in *EuroPLoP*, 1998, pp. 11–34.
31. J. Moreno and G. Robles, Automatic detection of bad programming habits in scratch: A preliminary study, in *IEEE Frontiers in Education Conf. Proc.*, 2014, pp. 1–4.
32. A. Elssamadisy and G. Schalliol, Recognizing and responding to "bad smells" in extreme programming, in *Proc. Int. Conf. Software Engineering*, 2002, pp. 617–622.
33. F. A. Fontana, M. V. Mäntylä, M. Zanoni and A. Marino, Comparing and experimenting machine learning techniques for code smell detection, *Empir. Softw. Eng.* **21**(3) (2016) 1143–1191.
34. C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall and G. Robles, On the usage of pythonic idioms, in *Proc. Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software*, 2018, pp. 1–11.
35. E. RodriguesJr. and R. Terra, How do developers use dynamic features? The case of Ruby, *Comput. Lang., Syst. Struct.* **53** (2018) 73–89.
36. P. Kyriakakis, A. Chatzigeorgiou, A. Ampatzoglou and S. Xinogalos, Exploring the frequency and change proneness of dynamic feature pattern instances in PHP applications, *Sci. Comput. Program.* **171** (2019) 1–20.
37. Z. Chen, W. Ma, W. Lin, L. Chen and B. Xu, Tracking down dynamic feature code changes against Python software evolution, *Int. Conf. Trustworthy Systems and Their Applications*, 2016, pp. 54–63.
38. B. Wang, L. Chen, W. Ma, Z. Chen and B. Xu, An empirical study on the impact of Python dynamic features on change-proneness, *Int. Conf. Software Engineering and Knowledge Engineering*, 2015, pp. 134–139.
39. L. Prechelt, An empirical comparison of seven programming languages, *Computer* **33**(10) (2000) 23–29.
40. P. Denny, A. Luxton-Reilly and E. Tempero, All syntax errors are not equal, in *Proc. 17th ACM Annual Conf. Innovation and Technology in Computer Science Education*, 2012, pp. 75–80.
41. Z. Ni, B. Li, X. Sun, T. Chen, B. Tang and X. Shi, Analyzing bug fix for automatic bug cause classification, *J. Syst. Softw.* **163** (2020) 110538.
42. A. Hammouri, M. Hammad, M. Alnabhan and F. Alsarayrah, Software bug prediction using machine learning approach, *Int. J. Adv. Comput. Sci. Appl.* **9**(2) (2018) 78–83.
43. M. Pradel and K. Sen, Deepbugs: A learning approach to name-based bug detection, *Proc. ACM Program. Lang.* **2** (2018) 1–25.
44. Y. Li, S. Wang, T. N. Nguyen and S. Van Nguyen, Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* **3** (2019) 1–30.
45. H. Zhang, L. Gong and S. Versteeg, Predicting bug-fixing time: An empirical study of commercial software projects, *Int. Conf. Software Engineering*, 2013, pp. 1042–1051.
46. M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White and D. Poshyvanyk, An empirical investigation into learning bug-fixing patches in the wild via neural machine translation, in *Proc. Int. Conf. Automated Software Engineering*, 2018, pp. 832–837.

47. B. Ray, D. Posnett, V. Filkov and P. Devanbu, A large scale study of programming languages and code quality in github, in *Proc. Int. Symp. Foundations of Software Engineering*, 2014, pp. 155–165.
48. Z. Chen, W. Ma, W. Lin, L. Chen, Y. Li and B. Xu, 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.* **61**(1) (2018) 012107.