Towards a Large-Scale Empirical Study of Python Static Type Annotations

Xinrong Lin Baojian Hua* Yang Wang* Zhizhong Pan
School of Software Engineering
University of Science and Technology of China
{lxr1210, sg513127}@mail.ustc.edu.cn {bjhua, angyan}@ustc.edu.cn*

***Abstract*—Python, as one of the most popular and important programming languages in the era of data science, has recently introduced a syntax for static type annotations with PEP 484, to improve code maintainability, quality, and readability. However, it is still unknown whether and how static type annotations are used in practical Python projects.**

This paper presents, to the best of our knowledge, the *first* and most *comprehensive* empirical study on the defects, evolution and rectification of static type annotations in Python projects. We first designed and implemented a software prototype dubbed PYSCAN, then used it to scan notable Python projects with diverse domains and sizes and type annotation manners, which add up to 19,478,428 lines of Python code. The empirical results provide interesting findings and insights, such as: 1) we proposed a taxonomy of Python type annotation-related defects, by classifying defects into four categories; 2) we investigated the evolution of type annotation-related defects; and 3) we proposed automatic defect rectification strategies, generating rectification suggestions for 82 out of 110 (74.55%) defects successfully. We suggest that: 1) Python language designers should clarify the type annotation specification; 2) checking tool builders should improve their tools to suppress false positives; and 3) Python developers should integrate such checking tools into their development workflow to catch type annotation-related defects at an early development stage.

We have reported our findings and suggestions to Python language designers, checking tool builders, and Python developers. They have acknowledged us and taken actions based on our suggestions. We believe these guidelines would improve static type annotation practices and benefit the Python ecosystem in general.

***Index Terms*—Empirical Study, Python, Static Type Annotations**

I. INTRODUCTION

Python continues to be one of the most popular and important programming languages in the era of data science [1] [2] [3]. As a dynamically typed language, Python allows rapid prototyping without type declaration or static type checking before execution. Although dynamic typing offers development flexibility, prior studies [4] [5] [6] [7], unfortunately, have demonstrated that dynamic typing may lead to potential and subtle type-related issues, which may have negative impacts on development productivity, code usability, and quality.

To mitigate this issue, Python has introduced a syntax for *static* type annotations with PEP 484 [8], to enable static checking before program execution and thus may detect type-related defects at an early development stage. Based on this language proposal, multiple third-party static type checking and inference technologies, as well as checking tools, have been proposed [9] [10] [11] [12] [13]. Although adding type annotations to Python may potentially benefit Python developers and projects, it is, unfortunately, still unknown whether and how static type annotations and checking tools are used in practice.

One may speculate that the study of static type annotation and checking tools is a solved problem, as there have been a significant amount of studies in this direction [7] [14] [15] [16] [17]. However, three issues still troubled developers: first, an official and authoritative static checking criterion is still absent. Type annotation, whose initial goal as specified by PEP 484 to ease static analysis, refactoring potential runtime type checking [8], is *optional* and *non-mandatory*. As a result, Python provides neither a criterion for static checking nor an official checking tool distributed with the official compiler. Hence, static checking of Python code currently relies heavily on third-party tools, whose effectiveness and reliability are still unknown.

Second, a unified taxonomy of defects for static type annotations is still lacking. Although existing studies have demonstrated that static checking tools (e.g. Mypy [9]) can help detect type annotation-related defects [6], different tools, unfortunately, detect, classify and report defects in different ways due to the lack of unified taxonomy, which may further bring confusions to end users [18].

Finally, it is challenging to rectify type annotation-related defects in Python programs in a timely and automatic manner. It is error-prone and time-consuming to rectify type annotation-related defects manually, especially for large Python projects. Worse yet, existing studies on automatic type-related rectification focus mostly on dynamic inconsistencies rather than static ones [17].

To this end, to study static type annotations in Python, several key questions remain unanswered: What is the taxonomy of type annotation-related defects? To what extent existing static type checking tools can detect these defects? How do type annotation and defects evolve and distribute in Python projects? Can these type annotation-related defects be rectified timely and automatically? Without such knowledge, Python

* Corresponding authors.

language designers might miss opportunities to further improve the language design, tool builders may build on incorrect assumptions, and Python developers might miss opportunities to improve code quality and reduce code maintenance costs by leveraging static type annotations and checking tools.

Our work. To answer the above key research questions, this paper presents, to the best of our knowledge, the *first* and most *comprehensive* empirical study of static type annotation practice in Python. This study is performed in several steps. First, to detect static type annotation-related defects, we designed and implemented a novel software prototype dubbed PYSCAN, by leveraging state-of-the-art Python checking tools.

Second, we selected and created a dataset of 13 notable Python projects with diverse domains and sizes, which add up to 19,478,428 lines of Python code. We have released our software prototype, dataset and empirical results.

Finally, to investigate the extent to which the information of defects provided by type checking tools can help the automatic generation of rectification suggestions, we utilized simple yet effective syntax-directed rectification strategies to fix type annotation-related defects in a timely and automated manner.

The empirical results provide interesting findings and insights, such as: 1) we proposed a taxonomy of Python type annotation-related defects, and classified defects into 4 categories: Inconsistent Type Annotation, Insufficient Safety Check, Incorrect Redefinition/Overload, and Wrong Type Annotation; 2) the proportion of functions with type annotations in well-maintained Python projects continues to increase; and 3) we proposed automatic defect rectification strategies, which are effective: PYSCAN successfully generate rectification suggestions for 82 out of 110 (74.55%) type annotation-related defects.

Based on the above empirical results, we suggest that: 1) Python language designers should clarify the type annotation specification; 2) checking tool builders should improve checking tools to suppress false positives; and 3) Python developers should integrate such checking tools into their development workflow to catch type annotation-related defects in an early stage. We have made our empirical data publicly available, reported our findings and suggestions to Python language designers, checking tool builders and Python developers. They have acknowledged us and taken actions based on our suggestions. We believe these guidelines would improve static type annotation practices and benefit the Python ecosystem in general.

Our findings, empirical results, tools and suggestions will benefit several audiences. Among others, they 1) provide suggestions to Python language designers to improve the static type annotation design; 2) help checking tool builders to further improve their tools and reduce false positives; and 3) help Python developers to discover and rectify type annotation-related defects in an early development stage.

Contributions. To the best of our knowledge, this work represents the first step toward a comprehensive empirical study of static type annotation-related defects in Python. To summarize, our work makes the following contributions:

- **Dataset.** We created a dataset of notable Python projects with diverse domains, sizes, and type annotation manners, which add up to 19,478,428 lines of Python code.
- **Empirical study and tools.** We presented the first and most comprehensive empirical study on Python static type annotation-related defects, with a novel software prototype we created dubbed PYSCAN.
- **Findings and insights.** We presented empirical results, findings and insights from the analysis, which benefited several audiences.
- **Open source.** We make our dataset, tool and empirical data publicly available in the interest of open science at `https://doi.org/10.5281/zenodo.7501198`.

Outline. The rest of this paper is organized as follows. Section II presents the background for this work. Section III presents the methodology for the study. Section IV presents empirical results by answering research questions. Section V and VI discusses implications for this work and threats to validity, respectively. Section VII discusses the related work, and Section VIII concludes.

II. BACKGROUND

To be self-contained, in this section, we present necessary background information on static/dynamic type systems (II-A) and type annotations (II-B).

A. Static/Dynamic Type Systems

A type system is a method of assigning types to variables, expressions, functions, and data structures in a program to ensure its type safety and maintainability. Type systems can be classified into two categories: static type systems and dynamic ones, according to whether type checking is performed at compile-time or at runtime [19]. Existing studies [20] [21] [22] have demonstrated that the two forms of type systems have their advantages and disadvantages, respectively. On the one hand, static type systems can capture type-related errors before program execution, improving the readability of the program and benefiting software maintenance [23]. On the other hand, dynamic type systems allow fast adaptation to requirement changes and rapid prototyping, without adding explicit type annotations [21]. Although the debate about the possible advantages and disadvantages of static or dynamic type systems in programming languages has been going on for a long time, both kinds of type systems play important roles with different capabilities and application scenarios.

B. Type Annotation

Prior studies [19] [20] [21] [22] [24] [20] [23] have demonstrated that adding optional static type annotations or performing type inference for dynamic programming language can improve developer productivity, code usability, code quality, and code readability. Dynamic programming languages such as TypeScript [25] and Ruby [26] have been embracing *gradual typing* [27], allowing programmers to control the degree of static checking by adding *optional* type annotations. "Optional" means that 1) developers can add type annotations

for part of code as desired; and 2) such type annotations have no effect on the runtime behavior of the target programs.

To benefit from static checking and to help early detection of defects and improve software maintenance, Python [28] has introduced a syntax for optional type annotations with PEP 484 [8] in 2015, allowing developers to add type annotations for function parameters, return values, variable initializations, and so on. For example, the following code snippet makes use of Python static type annotations. The parameter `x`, variable `z` and return type of the function `add` are annotated with the

```
1 def add(x: int, y) -> str:
2   z: int = x + y
3   return z
4   # Error: Incompatible return value type
```

types `int`, `int` and `str`, respectively, while the parameter `y` has no type annotations. Although static type annotations do not affect the execution of Python programs, they can be leveraged to perform static type checking by third-party tools (e.g., Mypy [9]) to detect type-related defects before execution. Taking the above code snippet as an example, Mypy will detect the type annotation defect caused by the inconsistency between the type of `z` and function return type of `add` at line 3. This defect can be rectified by changing line 1 to `def func(x: int, y) -> int:`. However, a unified taxonomy of static type annotation defects is still lacking. Worse yet, a systematic study of defect rectification has not been conducted before.

III. Methodology

In this section, we present the methodology to conduct the empirical study. It is challenging to perform an empirical study for large Python projects such as TensorFlow, for two key reasons: 1) automation: the study should be highly automatic, because it is time-consuming and error-prone to manually process thousands of defect reports generated by different checking tools; manual code inspection is only required when analyzing defect patterns and precision of tools; and 2) scalability: the study can be applied to any Python projects with different structures instead of specific ones.

To this end, we designed and implemented a novel software prototype dubbed PySCAN to detect and rectify static type annotation-related defects in Python projects, which are supplemented by human efforts to inspect specific defect patterns. We first introduce the architecture of PySCAN (Section III-A), then describe the design and implementation details of the frontend (Section III-B), the defect detection (Section III-C), the normalization (Section III-D), the automatic rectification suggestion (Section III-E), the validation (Section III-F), and the rectified program generation (Section III-G), respectively.

A. The Architecture

Two principles guide the architecture design of PySCAN. First, the architecture of PySCAN should be easily used to support the defect detection and rectification of different Python projects with easy configurations. Second, the architecture of

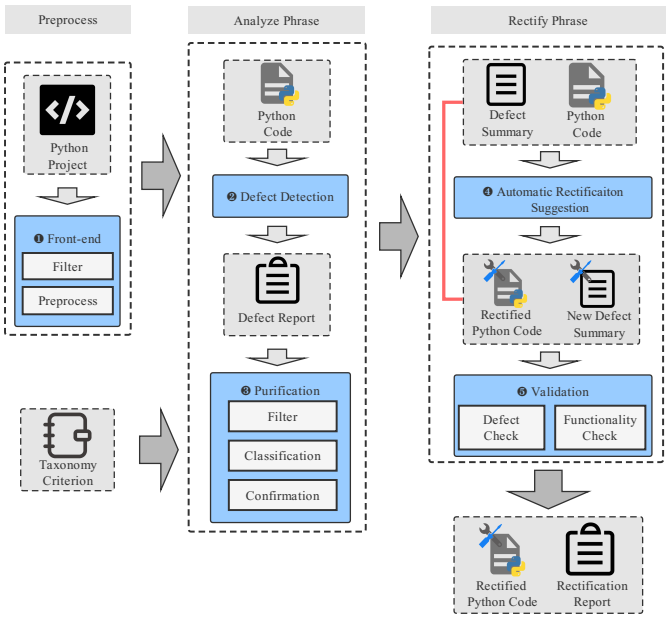

Fig. 1: The Architecture of PySCAN

PySCAN should be modular so that each module can be extended or replaced individually.

Based on the above principles, we present, in Fig. 1, the architecture of PySCAN, consisting of five key modules. First, the frontend module (❶) pulls target Python projects from data sources such as GitHub, filters out useless files and outputs Python sources. Second, the defect detection module (❷) analyzes the Python code in the target projects, and outputs original defect reports. Third, the normalization module (❸) classifies the original defect reports according to the taxonomy criterion (as presented in Section IV-E) and outputs a defect summary in a unified format. Fourth, the automatic rectification suggestion module (❹) takes as inputs both the original Python source code and the defect summary, generates corresponding rectification suggestions for Python developers' reference, and outputs a rectified Python source code. Finally, the validation module (❺) takes as inputs the original Python source, the rectified Python source and the corresponding new defect summary, to validate the effect of rectification suggestions and normal functionality of code, then outputs the rectified code, along with a detailed defect report.

In the following sections, we discuss the design and implementation details of each module, respectively.

B. The Frontend

Different Python projects organize source code and other files in different structures. Furthermore, besides Python source code, a Python project may contain supplementing files such as git configurations, scripts, tests, and documents.

To handle these peculiarities, the frontend normalizes the source files by: 1) filtering source Python code by removing components that are irrelevant to type annotation, such as documents and scripts; and 2) generating a list of Python files

to scan. Although it is possible to combine the frontend with other phases, the current design of PYSCAN, from a software engineering perspective, has two key advantages: 1) it makes it possible for PYSCAN to process Python projects with different structures easily; and 2) it makes PYSCAN more efficient to detect defects by removing irrelevant files at an early stage.

C. Defect Detection

To conduct an empirical study on static type annotation-related defects in Python, we need to detect defects in the first place. To this end, the defect detection module of PYSCAN leverages state-of-the-art static type annotation checking tools, to generate a detailed defect report for subsequent processing.

We have selected Python static checking tools according to one important criterion: the selected tools should be able to generate complete and detailed information, for the specific defect detected. This information should include, but not be limited to, error codes, file paths, line numbers, and so on. The reason is that this detailed information is indispensable in investigating to what extent the detected defects can be rectified in a fully automated manner (Section III-E). Without detailed defect information such as source locations, it is difficult, if not impossible, to rectify defects automatically.

Based on the above criterion, we have selected, as presented in TABLE I, four state-of-the-art and widely used static Python checking tools: Mypy [9], Pytype [13], Pyre [11], and Pyright [10]. All tools are actively maintained, open source, and developed following the typing standard PEP 484.

TABLE I: Static checking tools leveraged in this study.

Tools	Github Stars	Last Updated	First Release	License	Develop Lang.
Mypy[9]	14k	2022.10	2012	MIT	Python
Pytype[13]	3.9k	2022.10	2015	Apache v2.0	Python
Pyre[11]	6.5k	2022.10	2018	MIT	OCaml
Pyright[10]	8.5k	2022.10	2019	MIT	TypeScript

This work mainly investigates static type annotation-related defects in Python, for two key reasons: 1) prior studies [18] have demonstrated such defects are very common and serious to Python projects; and 2) state-of-the-art static checking tools are good at detecting such defects. However, it should be noted that our framework PYSCAN can easily leverage other checking tools as well.

D. Normalization

The normalization module takes as inputs the original defect reports generated by different checking tools along with the taxonomy criterion, and outputs a defect summary for subsequent processing.

Different type-checking tools might make use of different error codes even for the same defect [18]. To facilitate the subsequent automatic rectification and empirical study, the normalization module has two key functionalities: 1) classifying each original defect report generated by each tool into the corresponding category, according to the taxonomy criterion we proposed; and 2) extracting detailed defect information (e.g., file paths, line numbers) to generate defect summary in a unified format.

It should be noted that the presence of a normalization module does not demonstrate the limitation of any static checking tools. Instead, as different static checking tools focus on different categories of defects, detect and report defects in different ways, as they are designed with different goals and rationales, the normalization module resolves such discrepancies.

E. Automatic Rectification Suggestion

The automatic rectification suggestion module takes as inputs the defect summary from the normalization module as well as the original Python code, automatically generates rectification suggestion for the Python source code according to the defect summary for Python developers' reference. If the developers approve of a rectification suggestion generated by PYSCAN, the corresponding defects in the Python code were rectified automatically. The rectified Python code will be further validated by subsequent phases.

It should be noted that one primary goal of this study is to investigate to what extent the information of defects produced by checking tools can help generate effective rectification suggestions automatically, thus providing deeper insights into the nature of type annotation-related defects in Python. It supplements but does not substitute developer code reviews or existing code quality testing infrastructures such as CI/DI.

Our current proof-of-concept implementation of PYSCAN employs simple yet effective syntax-directed strategies to generate rectification suggestions for type annotation-related defects, from the defect summary.

F. Validation

Once the code is rectified, the validation module takes as input the original Python code, the rectified code and their corresponding defect summaries, performs two kinds of validations: 1) effect of defect rectifications; and 2) normal functionality of code. The criterion for determining a successful rectification is that defects are rectified without changing functionalities.

First, to validate the defect rectification effect, we employ a differential testing approach [29], by measuring the change of type annotation defect numbers. To be specific, PYSCAN compares the two defect summaries generated by the detection module and purification module for the original code and the rectified code, respectively, to check whether defects disappear in the rectified code.

Second, to guarantee the normal functionality of code is not affected, we employ a regression testing approach [30]. To be specific, PYSCAN makes use of the test suite distributed with the corresponding Python projects, to test the normal functionality of the rectified code is not changed. Our empirical results generated by PYSCAN demonstrated that this strategy is effective in practice (Section IV-H).

G. Rectified Program and Report Generation

After the rectification, PYSCAN generates as outputs the rectified code, as well as the corresponding final report for subsequent analysis.

IV. EMPIRICAL RESULTS

In this section, we present the empirical results by answering research questions.

A. Research Questions

By presenting the empirical results, we mainly investigate the following research questions:

RQ1: Defect Taxonomy. What is the taxonomy to classify type annotation-related defects?

RQ2: Effectiveness. Is PYSCAN effective in detecting type annotation-related defects in notable Python projects?

RQ3: Evolution. How do the type annotation-related defects evolve in Python projects?

RQ4: Rectification Suggestions. Is PYSCAN effective in rectifying type annotation-related defects in Python projects automatically by generating rectification suggestions?

RQ5: Practical Impact. Are the empirical study and results in this work useful to Python language designers, checking tool builders, and Python developers?

B. Experimental Setup

All the experiments and measurements are performed on a server with one 20-core physical Intel i7 CPU and 64 GB of RAM running Ubuntu 22.04.

C. Dataset

As TABLE II presents, we selected and created a dataset of 13 notable Python projects with diverse domains, sizes, and type-annotation manners (*inline type annotation* or *stub files*) [31], which add up to 19,478,428 lines of Python code. Two principles guided our selection of the dataset: 1) we selected the 13 Python projects with higher TSV scores [32], which is a metric combining watcher, contributor and community activity; and 2) we selected TensorFlow, one of the projects with high TSV scores, to study the evolution of type annotation-related defects. However, it should be noted that PYSCAN can be used to process other Python projects without any technical difficulties.

D. Evaluation Metrics

We use $precision$ and $recall$ to measure the accuracy (or effectiveness) of the tools. The definition of these two metrics is in equation 1.

$$precision = \frac{tp}{tp + fp} \quad recall = \frac{tp}{tp + fn} \tag{1}$$

In the equation, we use tp, fp, fn to denote true positives, false positives, and false negatives, respectively. We take the union of true positives from all checking tools as our ground truth. Precision measures the ratio of true positives to the result of a tool, and recall measures the ratio of true positives to the ground truth. A tool with high precision may have low recall, thus given the importance of both precision and recall, we also compute the $F1$ score according to equation 2.

$$F1\ score = \frac{2 \times precision \times recall}{precision + recall} \tag{2}$$

F1 score can reflect the overall accuracy of an analysis engine.

TABLE II: Dataset used in this study.

Project	TSV Score	Files	LoC (K)	Stars	Watchers	Contributors
Cpython[33]	49.93	2,045	910	39.2k	1,395	1,749
Ansible[34]	47.41	1,560	253	49.2k	2,009	6,427
Core[35]	38.20	8,780	1,511	44.6k	1,350	2,791
FastAPI[36]	36.54	800	58	34.1k	527	240
Scikit-learn[37]	35.26	915	350	46.6k	2,256	2,300
Django[38]	34.62	2,751	447	58.7k	2,332	2,548
TensorFlow[39]	32.64	2,894	1,102	70.7k	2,936	820
Pandas[40]	31.28	1,444	576	30.5k	1,119	2,731
Flask[41]	28.96	70	18	56.1k	2,239	726
Transformers[42]	28.00	1,813	798	48.9k	756	943
Faceswap[43]	27.89	231	71	37.8k	1,505	87
Scrapy[44]	26.61	338	55	41.2k	1,829	483
Requests[40]	23.98	35	11	45.7k	1,393	694

TABLE III: Summary of the taxonomy of the type annotation-related defects.

Taxonomy	**Inconsistent Type Annotation**	**Insufficient Safety Check**	**Incorrect Redefinition / Overload**	**Wrong Type Annotation**
Patterns	Inconsistent Return Type	Insufficient Checks for None	Multiple Definitions	Undefined or Invalid Type
	Inconsistent Parameter Type	Insufficient Checks for Existence of Members	No Applicable Overload or Redefinition	Illegal Annotation Target
	Inconsistent Variable Type	Insufficient Checks for Operator Support	-	-
	Inconsistent Attribute Type	-	-	-

E. RQ1: Defect Taxonomy

To answer **RQ1** by presenting a taxonomy of the type annotation-related defects, we first apply PYSCAN to the latest versions of projects in the dataset to obtain the original results. We then create a taxonomy of type annotation-related defects, as shown in TABLE III, based on the description and pattern of defects, following an inductive coding approach [45].

Due to space limitations, we only present representative defect patterns here, all original results are released in our replication package.

1) **Inconsistent Type Annotation:** Inconsistent Type Annotation (ITA) denotes the actual types of target variables are inconsistent with the types annotated. Such inconsistency may mislead programmers and affect code maintainability. We identified 4 specific patterns of ITA and describe each of them below, respectively.

TABLE IV: Distribution of defects in sample sets and original corpus.

Tool	#Sample set / #Original corpus				
	ITA	ISC	IRO	WTA	Total
Mypy	6/839	7/606	2/145	0/19	15/1,609
Pyre	0/64	0/0	2/253	160/21,371	162/21,688
Pyright	76/10,262	97/13,524	0/163	0/0	173/23,949
Pytype	8/1,015	33/4,700	0/0	9/1,384	50/7,099
Total	90/12,180	137/18,830	4/561	169/22,774	400/54,345

a) Inconsistent Return Type: the actual type of the function's return value is inconsistent with the type annotated in the function definition. For instance, Pyre reported a defect for the incompatible return type `None` at line 4 in the following code snippet, where the annotation at line 2 indicates that the

```
1 def _get_account(account_identifier: str) ->
2   IcloudAccount:
3   if account_identifier is None:
4     return None
```

expected return type of function `_get_account` is `IcloudAccount`, leading to inconsistency.

b) Inconsistent Parameter Type: the types of actual function arguments are inconsistent with the annotated parameter types. For example, Pyright reported a defect for the incompatible parameter type at line 7 in the following code snippet, where

```
1 def drop_path(...,drop_prob:float=0.0,...):
2 class ConvNextDropPath(nn.Module):
3   def __init__(self,
4   drop_prob:Optional[float]=None)->None:
5     self.drop_prob = drop_prob
6   def forward(...) -> ...:
7     return drop_path(...,self.drop_prob,...)
```

the type of `drop_prob` (`Optional[float]`) is inconsistent with the declared type `float` (at line 1).

c) Inconsistent Variable Type: the type a variable being explicitly annotated is inconsistent with its actual type.

d) Inconsistent Attribute Type: the type assigned to an attribute is inconsistent with the actual type of the attribute.

In our released open source, we include examples for these two taxonomy, but omit them here for space considerations.

2) **Insufficient Safety Check:** Insufficient Safety Check (ISC) represents the lack of necessary safety check for unexpected type, which might lead to unexpected errors in subsequent program execution. We identified 3 specific patterns of ISC and describe each of them below, respectively.

a) Insufficient Checks for `None`: a variable may be assigned `None` during execution, which may lead to unexpected behaviors if developers failed to handle correctly. For example, Pyright reported a defect at line 3 in the following code snippet. The variable `match` at line 2 may be assigned a

```
1 netloc_re = re.compile(...)
2 match = netloc_re.match(parts[1])
3 auth = match.group(1)
```

`None` value; hence, subsequent accesses to `match` (line 3) may trigger runtime type errors.

b) Insufficient Checks for Existence of Members: this error will trigger runtime failures, when a program accesses a method or attribute that is absent from the target class or module.

c) Insufficient Checks for Operator Support: this type of error is triggered when the operator does not support specific types of operands. For instance, Mypy reported a defect at line 2 in the following code snippet. The second "+" operator adds a variable `age` of type `int` to a value of type `str`, which will trigger a type error at runtime.

```
1 def get_name_with_age(name: str, age: int):
2   name_with_age=name+" is this old: "+age
3   return name_with_age
```

3) **Incorrect Redefinition/Overload:** Incorrect Redefinition/Overload (IRO) represents a variable is redefined during its lifetime. Although dynamically typed programming languages such as Python allow redefinition and overloading for rapid prototyping, such practices lead to maintainability issues with project evolution. We have identified two specific defect patterns, which are described below.

a) Multiple Definitions: Modules, classes or functions can be redefined in a single namespace, where the latter definition overwrites the preceding one. Multiple definitions might defeat checking tools which failed to determine the correct definitions. As an example, Mypy reported a defect at line 4 in the following code snippet, where the variable `ints_to_pydatetime` overwrites a variable with the same name but imported from `tslibs` instead of `tslib`.

```
1 try: from ..tslibs import ints_to_pydatetime
2 except ImportError:
3   from ..tslib import ints_to_pydatetime
```

b) No Applicable Overload or Redefinition: This category of errors is triggered, when a function is called but none of the signatures matches the argument types.

4) **Wrong Type Annotation:** Wrong Type Annotation (WTA) represents that annotations are not used correctly by developers, either at wrong places or with wrong syntax. WTA not only causes type safety issues, but also may lead to runtime errors. We identified two common patterns of WTA and describe each of them below, respectively.

a) Undefined or Invalid Type: This type of error occurs when invalid types or type aliases are used as annotations. For example, Pyre reported a defect at line 2 in the following code snippet, where the type of `chinese_word_set` should be annotated `Set[int]` instead of `set()`.

TABLE V: Defects detected by PYSCAN in the sample set.

Tool	ITA		ISC		IRO		WTA		Total				
	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Precision	Recall	F1
Mypy	6	3	7	3	2	2	0	0	15	8	53.33%	7.27%	12.80%
Pyre	0	0	0	0	2	1	160	15	162	16	9.88%	14.55%	11.76%
Pyright	76	37	97	45	0	0	0	0	173	82	47.40%	74.55%	57.95%
Pytype	8	3	33	0	0	0	9	1	50	4	8.00%	3.64%	5.00%
Total	90	43	137	48	4	3	169	16	400	110	/	/	/

```
1 def add_sub_symbol(bert_tokens: List[str],
2   chinese_word_set: set()):...
```

b) Illegal Annotation Target: A type annotation is applied to an illegal target. For example, in Python, re-annotating a previously annotated target or annotating a target after its first declaration, is illegal. For example, Pyre reported a defect at line 3 of the following code snippet, as annotating `model` a type `GenerativeQAModule` violates its initialization with the constant `None` at line 1.

```
1 def main(args=None, model=None) -> ...:
2   if model is None:
3     model: GenerativeQAModule = ...
```

Summary: We proposed a taxonomy to classify type annotation defects into 4 categories, namely Inconsistent Type Annotation (ITA), Insufficient Safety Check (ISC), Incorrect Redefinition/Overload (IRO), and Wrong Type Annotation (WTA).

F. RQ2: Effectiveness

To answer **RQ2** by investigating the effectiveness of PYSCAN, we study the distribution of various categories of type annotation-related defects.

As prior studies [18] demonstrated, static checking tools may have high false positives. Our empirical study utilized 4 state-of-the-art tools, as presented in TABLE I, to scan the 13 real-world representative Python projects, resulting in an original corpus with 54,345 defect reports. It is a great challenge to analyze all these reports manually. Hence, following the sampling guideline proposed by Krejcie and Morgan [46], to reach a 95.0% confidence level and a 5.0% confidence interval, we selected a representative sample set with sizes of 381, from 54,345 original defect reports, and rounded up it to 400 for convenience. We then conduct a manual inspection of the reports. TABLE IV presents the distribution of different categories of defects in the sample set along with the original corpus, which demonstrates an insignificant difference between the sample set and original corpus in the distribution of defects.

To calculate the precision of each static checking tool, we formed an inspection group with 3 graduate students who are familiar with Python, type annotation, and checking tools, to independently conduct a manual inspection of all original reports in the sample set generated by the 4 static checking tools. Moreover, to ensure the reliability of the inspection results, we adopted the Fleiss' Kappa statistic [47], which is frequently used to test inter-rater reliability. The inspection resulted in a Fleiss' Kappa score of 0.892, which indicates an "Almost Perfect" agreement. In the rare cases where the students disagreed, we conservatively judged these reports as false positives.

TABLE V presents the empirical results: the first column gives the names of projects. The next 4 columns present the numbers of defects detected by 4 static checking tools as listed in TABLE I, respectively. According to the defect taxonomy we presented in **RQ1**, for each of the 4 categories: Inconsistent Type Annotation (ITA), Insufficient Safety Check (ISC), Incorrect Redefinition/Overload (IRO) and Wrong Type Annotation (WTA), we presented the number of vulnerabilities detected by each tool, as well as true positives (TPs) among them. The last 4 columns present the total true positives, as well as the metrics of precision, recall, and F1, respectively.

The empirical results give interesting findings and insights. First, although Pyre and Pytype have a low precision (under 10%), Mypy and Pyright reported defects more precisely. Among them, Mypy has both the highest precision (53.33%) and relatively low recall (7.27%), which leads to its low F1 score (only 12.80%), while Pyright has both the highest recall (74.55%) and F1 score (57.95%).

Second, to further investigate the root causes of false positives, the inspection group of students conduct a manual source code inspection of the corresponding source code. This inspection revealed 2 key reasons leading to false positives: 1) limited code context: for instance, Pyright reported an Insufficient Safety Check (ISC) defect for the target `self.tokenizer` at line 1 in the following code snippet. However, the `try/exception` block can guarantee that the exception will be caught, even when `self.tokenizer` is `None`; and 2) failing to identify type name: for example, we have identified that 145 of the 160 Wrong Type Annotations (WTA) reported by Pyre in TABLE V are caused by the failure to recognize type aliases and class names imported from other modules. Fortunately, most checking tools are still under active development and their future improvements might suppress such false positives.

```
try: vocab = self.tokenizer.get_vocab()
except Exception: vocab = {}
```

Third, as TABLE IV and V presented, the 4 checking tools have different detection capabilities, which are caused by the different design rationales and goals of these tools. Mypy [9], co-developed with PEP 484, only focuses on code that is explicitly annotated, thus leading to relatively few bug reports. Pyre [11] focuses on the correctness of type annotations. Pyright [10] is more aggressive and strict at the safety checking as well as the consistency of type annotations, which is reflected in its largest number of defect reports. We speculate the reason is that as Pyright is used as one of the official type checking plugins in VSCode, one of its key design goals is to help developers improve their code quality as much as possible. Pytype [13] can perform type inference for variables without type annotation, but it does not focus on variable redefinitions and overloads, which are related to the "Pythonic" programming style of adding type annotations without affecting runtime behaviors.

Summary: Notable Python projects do contain considerable type annotation defects. And state-of-the-art type checking tools have different detection capabilities, due to their different design rationales and goals, in which Pyright is superior to other checking tools with the highest F1 score.

G. RQ3: Evolution

To answer **RQ3** by investigating the evolution of static type annotations and defects, we applied PYSCAN to 12 versions (from 2.0.0 to the latest 2.11.0) of TensorFlow [40]. We selected TensorFlow, as it is a representative Python project with a high TSV score of 32.64. The start version is set to 2.0.0 (released in 2019), as versions before that do not make use type annotations, even though PEP 484 [8] was released in 2015. TABLE VI presents the versions, release time, distribution of defects, number of Python functions (#Func), number of functions with type annotations (#Func_TA) and the ratios to all functions (TA_ratio), the numbers of defects per function with type annotations (DPTF), line of code (LoC), and the numbers of defects per thousand lines (DPTL), in 12 versions of TensorFlow.

Several interesting findings and insights can be obtained from the empirical results. First, the ratio of functions with type annotations increased from 4.99% in 2019 to 39.12% in 2022 (that is, about 1 function in every 3 functions is annotated with types). We speculate the reason for this increase is that developers pay more attention to type annotations to acknowledge its software engineering advantages.

Second, to gain a deeper understanding of how the overall code quality evolves, we calculated the numbers of defects per function with type annotations (DPTF), as well as the numbers of defects per thousand lines of code (DPTL), which are well-established metrics for measuring code quality [48] [49] [50]. From 2019 to 2022, the DPTF of TensorFlow decreased from 708.50 to 34.08, and the DPTL decreased from 15.06 to 11.99, which indicates that the code quality, from the type safety perspective, is continuously improving with the introduction of type annotations. We identified 3 reasons for the code quality improvement: 1) the specification of Python type annotations are improved since its first release [51] [52] [53]; 2) Python developers' understanding of type annotations keeps deepening; and 3) improvement of static type checking tools is effective in detecting and fixing type annotation defects in a timely manner.

Finally, in all 12 versions of TensorFlow, Insufficient Safety Check (ISC) defects constitutes a major proportion (over 70%) of all defects, which is on par with our finding in RQ2.

Summary: Both the increase of type-annotated function ratios (from 4.99% in 2019 to 39.12% in 2022) and the declines in DPTF and DPTL indicate that the code quality of TensorFlow improves gradually. Three reasons account for this improvement: type annotation specification evolution, checking tools improvement, and developers' deeper understanding of type annotations. Insufficient Safety Check defects constitute the largest portion of all defects (over 70%).

H. RQ4: Rectification Suggestion

To answer **RQ4** by investigating to what extent the information of defects provided by checking tools can help PYSCAN generate rectification suggestions automatically, we apply PYSCAN to the 110 TPs from the sample set (Section IV-F). The inspection group of students conducted a manual inspection of all rectification suggestions generated for the 110 defects by PYSCAN to check whether the rectification suggestions are successfully generated. The results are presented in TABLE VII.

We obtain several important findings from these empirical results. First, the automatic rectification is quite effective. As TABLE VII presents, PYSCAN successfully generated rectification suggestions for 82 out of 110 (74.55%) type annotation-related defects in total, including 36 ITAs (83.72%), 44 ISCs (91.67%), 1 IRO (33.33%), and 1 WTA (6.25%), in a fully automatic manner.

Second, to understand why PYSCAN failed to generate rectification suggestions for some defects (especially IRO and WTA), we further manually analyzed all the corresponding code segments of defects that PYSCAN failed to rectify. This analysis revealed 2 key reasons for the failures: 1) some defect reports gave inaccurate locations, leading to incorrect rectification suggestions for ITA and ISC defects; and 2) our rectification strategies are syntax-directed and local and thus conservative: they cannot handle some complex syntax structures such as function redefinitions. However, our results for automatic type annotation rectification is still impressive, providing a starting point for potential Python IDE integration.

TABLE VI: Evolution of type annotations and type annotation-related defects, in 12 versions of TensorFlow.

Versions	Time	Type annotation-related defects					#Func	#Func_TA	TA_ratio	DPTF	LoC	DPTL
		ITA	ISC	IRO	WTA	Total						
2.0.0	2019.09	2,743	14,561	17	1,100	18,421	521	26	4.99%	708.50	1,223,333	15.06
2.1.0	2020.01	1,916	7,327	13	819	10,075	505	26	5.15%	387.50	921,081	10.94
2.2.0	2020.05	2,427	8,284	15	869	11,595	437	26	5.95%	445.96	983,245	11.79
2.3.0	2020.07	2,245	8,973	14	787	12,019	540	25	4.63%	480.76	1,050,116	11.45
2.4.0	2020.12	2,913	9,557	17	863	13,350	612	46	7.52%	290.22	1,106,322	12.07
2.5.0	2021.05	3,162	10,054	19	947	14,182	695	120	17.27%	118.18	1,138,525	12.46
2.6.0	2021.08	2,608	10,702	20	1,010	14,340	737	125	16.96%	114.72	1,165,060	12.31
2.7.0	2021.11	3,093	10,636	19	1,044	14,792	756	148	19.58%	99.95	1,162,901	12.72
2.8.0	2022.02	2,749	10,098	19	1,061	13,927	786	182	23.16%	76.52	1,124,542	12.38
2.9.0	2022.05	2,830	10,768	19	1,116	14,733	876	272	31.05%	54.17	1,139,710	12.93
2.10.0	2022.09	2,003	9,715	17	1,134	12,869	962	366	38.05%	35.16	1,096,436	11.74
2.11.0	2022.11	2,439	9,702	17	1,134	13,292	997	390	39.12%	34.08	1,108,668	11.99

TABLE VII: Rectification suggestions PYSCAN generated successfully.

Defect Category	#Detected	#Succss suggestions	Success ratio
Inconsistent Type Annotation	43	36	83.72%
Insufficient Safety Check	48	44	91.67%
Incorrect Redefinition/Overload	3	1	33.33%
Wrong Type Annotation	16	1	6.25%
Total	110	82	74.55%

Summary: PYSCAN successfully generated rectification suggestions for 74.55% type annotation-related defects automatically, demonstrating that most type annotation-related defects, especially Inconsistent Type Annotation and Insufficient Safety Check defects, can be rectified by a simple heuristic-based rectification strategy.

I. RQ5: Practical Impact

To answer **RQ5** by investigating the practical impact of PYSCAN, we sent our initial empirical results to three potential audiences: Python language designers, checking tool builders, and Python developers.

Python Language Designers. The Python language designers carefully reviewed and discussed our suggestions[1]. They have acknowledged us and taken actions based on our suggestions. The Python core designers affirmed that existing tools "are all mature and get the job done" and thus there is no need to develop an *official* Python checker. Although they admitted "I am personally partial to Mypy, which I helped co-develop for many years", they also indicated that "users have diverse needs, and one size does not fit all".

Interestingly, the Python language designers have similar findings as our work: 1) different tools sometimes give different results due to "different goals and audiences in mind, different limitations, different implementation, and so on."; and 2) they also admitted that "PEP 484 is not very strictly specified (we had little experience in the matter at the time).", leaving room open for the interpretation of type annotations and development spaces of various checking tools.

[1]https://discuss.python.org/t/is-it-possible-to-add-a-new-compile-option-for- static-type-checking-or-provide-an-official-type-checker/19945

Checking Tool Builders. The checking tool builders carefully considered our suggestions of making checking tools easier to integrate into CI workflows, which is also a feature many users have been expecting[2], and have taken corresponding actions to improve their tools.

Python Developers. Python project developers carefully reviewed the issues we reported, and acknowledged us. Among them, some Python developers admitted that they mainly use type annotations for documentation purposes, but have not intend to improve code quality[3] [4], while other developers believe that type annotations are of good value, even a best practice in some cases[5].

Summary: Python language designers, checking tool builders, and Python developers have all acknowledged us and have taken actions based on our suggestions.

V. IMPLICATIONS

This paper presents the first empirical study of static type annotation defects in Python. In this section, we discuss some implications of this work, along with some important directions for future research.

Python Language Designers. The results in this work provide Python language designers with important insights to further improve the Python type annotation specification. We suggest that Python language designers might: 1) clarify the type annotation specification based on the results in this study; 2) create an official and authoritative type checking tool, by either leveraging off-the-shelf static checking tools or developing a novel one; and 3) determine and release a unified taxonomy for type annotation defects, which will provide a guideline for improving checking tools.

Checking Tool Builders. The results in this work provide insights to tool builders to further improve these tools. In particular, answers to RQ2 demonstrated that state-of-the-art checking tools generated considerable false positives, which is laborious and time-consuming to inspect manually. We suggest

[2]https://github.com/python/mypy/issues/13874

[3]https://github.com/huggingface/transformers/issues/19515

[4]https://github.com/ansible/ansible/issues/79115

[5]https://mail.python.org/archives/list/python-ideas@python.org/thread/CG2IGZSV2Z4YMKLPK5MBWK4K4CDYTAFB/

that checking tool builders should: 1) put more research efforts into suppressing false positives; 2) improve their tools by providing informative rectification suggestions; and 3) extend the capability of their tools to detect more categories of type annotation-related defects.

Python Developers. To take its full advantage, Python developers have been embracing static type annotations [7] [6] [18]. However, the results of RQ2 and RQ3 demonstrated that even popular and well-maintained Python projects still contain considerable type annotation defects. Fortunately, these defects can be effectively detected by state-of-the-art static type checking tools. We suggest that Python developers can: 1) incorporate type annotations into their projects to improve readability and maintainability; and 2) leverage static checking tools into their development workflow to detect type-related defects in an early development stage.

VI. Threats to Validity

As in any empirical study, there are threats to validity with our work. We attempt to remove these threats where possible, and mitigate the effect when removal is not possible.

Dataset. In this work, we have used 13 Python projects with higher TSV scores [32] and 12 versions of TensorFlow as our dataset. In the meanwhile, there are other Python datasets in wild [18]. Fortunately, the modular design of PySCAN make it straightforward to study other Python projects, and we have made our tool and dataset open source and publicly available, which can be utilized to perform other studies.

Recall Computation. In this work, we compute recalls with respect to a ground truth of the union of all TPs reported by all checking tools. While the ground truth is reliable, they may be unfair to some checking tools, as different checking tools have different design rationales and detection goals, and thus focus on certain categories of defects. For example, while Pyre focuses on IRO and WTA, Mypy can detect all categories of defects. To mitigate this threat, we plan to create a manually crafted dataset as ground truth to compare different checking tools in a more comprehensive manner.

Classification. In this work, we present a taxonomy to classify all type annotation-related defects reported by four off-the-shelf and state-of-the-art static checking tools, into 4 categories. One threat here is that some specific defects may be classified into incorrect categories, due to the lack of a uniform taxonomy between these checking tools. To mitigate this threat, our inspection group has put considerable efforts into creating correction classification, and has inspected the sources of these checking tools when necessary.

Static Checking Tools. In this work, we used 4 static checking tools to perform this study. Then we take the union of all TPs identified as our ground truth. In the meanwhile, there are other checking tools (e.g., PySonar2 [54]) which may detect other defects, but we do not have the necessary resources to explore. Fortunately, the modular architecture of PySCAN (Fig. 1) is neutral to the specific checking tools used, thus it is easy to incorporate other checking tools in PySCAN.

VII. Related Work

In recent years, there have been a significant amount of studies on the Python type annotations. However, the work in this paper represents a novel contribution to this field.

Type Inference. There have been many studies on type inference. Mir et al. [14] presented Type4Py, a deep similarity learning-based hierarchical neural network model, to discriminate between similar and dissimilar types in a high-dimensional space. PYInfer [15] is an end-to-end learning-based type inference tool that automatically generates type annotations for Python variables. Ivanov et al. [16] took a hybrid approach to type prediction for Python. Wei et al. [55] proposed a probabilistic type inference scheme for TypeScript based on a graph neural network. Hu et al. [56] presented PYCTYPE to infer type signatures for foreign functions. There are also many checking tools to detect type misuses based on type annotations [9] [10] [11].

Our empirical study extends previous work in two ways: 1) we compare the capability of various tools in detecting defects, and proposed a taxonomy of static type annotation-related defects by using an inductive coding approach [45]; and 2) we analyzed potential reasons for false positives.

Empirical Study of Type Annotation Practices. There has been considerable work on type annotation practices. Jin et al. [31] defined six patterns of type annotation practices and revealed three complementary features of type-annotated files. Khan et al. [6] studied the extent to which Python projects benefit from type checking features. Chen et al. [7] provided empirical evidence of the relationships between dynamic typing related practices and bugs. Rak-amnouykit et al. [18] presents a study of Python3 type usage. Campora et al. [17] presented PyHound for detecting and fixing inconsistencies in Reticulated Python.

However, a major limitation of existing work is that they only discussed the usage of type annotation and the dynamic typing related practices, whereas our study focused on defects, evolutions, and rectifications of static type annotations.

VIII. Conclusion

In this work, we present the first empirical study of Python static type annotation practice, by utilizing a novel tool PySCAN. The empirical results show that the static type annotation defects can be rectified by leverage defect reports, and type annotations and checking tools can improve the Python code quality. Our findings can benefit several audiences including Python language designers, checking tool builders, and Python developers, and thus significantly expand the scope of existing studies of type annotations.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

References

[1] "Stack overflow developer survey 2022," https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022.

[2] "Tiobe: The big 4 languages keep increasing their dominance," https://www.tiobe.com/tiobe-index/.

[3] "Top programming languages 2022." https://spectrum.ieee.org/top-programming-languages-2022, Aug. 2022.

[4] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 1–18.

[5] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong China: ACM, Nov. 2014, pp. 155–165.

[6] F. Khan, B. Chen, D. Varro, and S. Mcintosh, "An empirical study of type-related defects in python projects," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[7] 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 *Proceedings of the 28th International Conference on Program Comprehension*. Seoul Republic of Korea: ACM, Jul. 2020, pp. 83–93.

[8] "Pep 484: Type hints," https://peps.python.org/pep-0484/.

[9] "Mypy: Optional static typing for python," http://mypy-lang.org/.

[10] "Pyright: Static type checker for python," Microsoft, Jul. 2022.

[11] "Pyre: A performant type-checker for python 3." https://pyre-check.org/.

[12] "Pylint: A static code analyser for python," https://pylint.pycqa.org/en/latest/.

[13] "Pytype: A static type analyzer for python code," https://google.github.io/pytype/.

[14] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Practical deep similarity learning-based type inference for python," in *Proceedings of the 44th International Conference on Software Engineering*, May 2022, pp. 2241–2252.

[15] S. Cui, G. Zhao, Z. Dai, L. Wang, R. Huang, and J. Huang, "Pyinfer: Deep learning semantic type inference for python variables," no. arXiv:2106.14316, Jun. 2021.

[16] V. Ivanov, V. Romanov, and G. Succi, "Predicting type annotations for python using embeddings from graph neural networks:," in *Proceedings of the 23rd International Conference on Enterprise Information Systems*. SCITEPRESS - Science and Technology Publications, 2021, pp. 548–556.

[17] J. P. Campora and S. Chen, "Taming type annotations in gradual typing," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 191:1–191:30, Nov. 2020.

[18] I. Rak-amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby, "Python 3 types in the wild: A tale of two type systems," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. Virtual USA: ACM, Nov. 2020, pp. 57–70.

[19] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in javascript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Buenos Aires: IEEE, May 2017, pp. 758–769.

[20] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, Jun. 2012, pp. 153–162.

[21] A. Stuchlik and S. Hanenberg, "Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time," in *Proceedings of the 7th Symposium on Dynamic Languages*, ser. DLS '11. New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 97–106.

[22] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, Oct. 2014.

[23] S. Spiza and S. Hanenberg, "Type names without static type checking already improve the usability of apis (as long as the type names are correct): An empirical study," in *Proceedings of the 13th International Conference on Modularity*. Lugano Switzerland: ACM, Apr. 2014, pp. 99–108.

[24] J. Siek and W. Taha, "Gradual typing for objects," in *ECOOP 2007 – Object-Oriented Programming*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and E. Ernst, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4609, pp. 2–27.

[25] "Typescript: Javascript with syntax for types." https://www.typescriptlang.org/.

[26] "Ruby: A dynamic, open source programming language with a focus on simplicity and productivity." https://www.ruby-lang.org/en/.

[27] M. Greenberg, "The dynamic practice and static theory of gradual typing," in *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[28] "Python: A programming language that lets you

work quickly and integrate systems more effectively," https://www.python.org/.

[29] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[30] W. Wong, J. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings The Eighth International Symposium on Software Reliability Engineering*, Nov. 1997, pp. 264–274.

[31] W. Jin, D. Zhong, Z. Ding, M. Fan, and T. Liu, "Where to start: Studying type annotation practices in python," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia: IEEE, Nov. 2021, pp. 529–541.

[32] "The two sigma ventures open source index," https://twosigmaventures.com/open-source-index/.

[33] "Cpython: The python programming language," https://github.com/python/cpython.

[34] "Ansible: Ansible is a radically simple it automation platform that makes your applications and systems easier to deploy and maintain." https://github.com/ansible/ansible.

[35] "Core: Open source home automation that puts local control and privacy first." https://github.com/home-assistant/core.

[36] "Fastapi: Fastapi framework, high performance, easy to learn, fast to code, ready for production," https://github.com/tiangolo/fastapi.

[37] "Scikit-learn: Scikit-learn: Machine learning in python," https://github.com/scikit-learn/scikit-learn.

[38] "Django: The web framework for perfectionists with deadlines." https://github.com/django/django, Oct. 2022.

[39] "Tensorflow: Large-scale machine learning on heterogeneous systems." https://github.com/tensorflow/tensorflow, Nov. 2015.

[40] "Pandas: Flexible and powerful data analysis / manipulation library for python." https://github.com/pandas-dev/pandas, Oct. 2022.

[41] "Flask: The python micro framework for building web applications." https://github.com/pallets/flask, Oct. 2022.

[42] "Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax." https://www.aclweb.org/anthology/2020.emnlp-demos.6, pp. 38–45, Oct. 2020.

[43] "Faceswap: Deepfakes software for all." https://github.com/deepfakes/faceswap.

[44] "Scrapy: A fast high-level web crawling & scraping framework for python." https://github.com/scrapy/scrapy, Oct. 2022.

[45] X. Huang, H. Zhang, X. Zhou, M. A. Babar, and S. Yang, "Synthesizing qualitative research in software engineering: A critical review," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 1207–1218.

[46] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities - robert v. krejcie, daryle w. morgan, 1970," *Educational and psychological measurement*, vol. 30, no. 3, pp. 607–610, 1970.

[47] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological Bulletin*, vol. 76, no. 5, pp. 378–382, Nov. 1971.

[48] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *Computational Science and Its Applications — ICCSA 2003*, ser. Lecture Notes in Computer Science, V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, Eds. Berlin, Heidelberg: Springer, 2003, pp. 724–732.

[49] Z. Shams and S. H. Edwards, "An experiment to test bug density in students' code (abstract only)," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: Association for Computing Machinery, Mar. 2013, p. 742.

[50] T. Bach, A. Andrzejak, R. Pannemans, and D. Lo, "The impact of coverage on bug density in a large industrial software project," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov. 2017, pp. 307–313.

[51] "Pep 526: Syntax for variable annotations," https://peps.python.org/pep-0526/.

[52] "Pep 585:type hinting generics in standard collections," https://peps.python.org/pep-0585/.

[53] "Pep 677: Callable type syntax," https://peps.python.org/pep-0677/.

[54] "Pysonar2: A semantic indexer for python with interprocedural type inference," Oct. 2022.

[55] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," *arXiv preprint arXiv:2005.02161*, 2020.

[56] M. Hu, Y. Zhang, W. Huang, and Y. Xiong, "Static type inference for foreign functions of python," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2021, pp. 423–433.