An Empirical Study of Type-Related Defects in Python Projects

Faizan Khan, *Student Member, IEEE,* Boqi Chen, *Student Member, IEEE,* Daniel Varro, *Member, IEEE* and Shane McIntosh, *Member, IEEE*

Abstract—In recent years, `Python` has experienced an explosive growth in adoption, particularly among open source projects. While `Python`'s dynamically-typed nature provides developers with powerful programming abstractions, that same dynamic type system allows for type-related defects to accumulate in code bases. To aid in the early detection of type-related defects, type annotations were introduced into the `Python` ecosystem (i.e., PEP-484) and static type checkers like `mypy` have appeared on the market. While applying a type checker like `mypy` can in theory help to catch type-related defects before they impact users, little is known about the real impact of adopting a type checker to reveal defects in `Python` projects.

In this paper, we study the extent to which `Python` projects benefit from such type checking features. For this purpose, we mine the issue tracking and version control repositories of 210 `Python` projects on GitHub. Inspired by the work of Gao et al. on type-related defects in JavaScript, we add type annotations to test whether `mypy` detects an error that would have helped developers to avoid real defects. We observe that 15% of the defects could have been prevented by `mypy`. Moreover, we find that there is no significant difference between the experience level of developers committing type-related defects and the experience of developers committing defects that are not type-related. In addition, a manual analysis of the anti-patterns that most commonly lead to type-checking faults reveals that the redefinition of `Python` references, dynamic attribute initialization and incorrectly handled Null objects are the most common causes of type-related faults. Since our study is conducted on fixed public defects that have gone through code reviews and multiple test cycles, these results represent a lower bound on the benefits of adopting a type checker. Therefore, we recommend incorporating a static type checker like `mypy` into the development workflow, as not only will it prevent type-related defects but also mitigate certain anti-patterns during development.

Index Terms—Software Defects, Static Type Checkers, Dynamic Type Systems, Empirical Study.

1 Introduction

Dynamically-typed languages are popular nowadays. They are easy to learn and allow programmers to rapidly prototype solutions [28], [37]. Yet, as the complexity of the codebases increase and developers work collaboratively, undisciplined use of dynamic language features can hinder the comprehensibility of a codebase, and in turn, hinder the productivity of software engineers. Even basic questions, such as "What is the return type of this function?" or "What type does this function expect?" can require a thorough exploration of the codebase to be concretely answered.

Despite the absence of a static type system, type checking features are now available in dynamically-typed languages through the use of *static type annotations*. With the emergence of language standards that facilitate annotations (e.g. PEP-484 for *Python*), type annotation tools like `mypy` (*Python*) and `flow` (*Javascript*) have found their way into the market.

While type checking can, in theory, prevent type-related defects, little is known about their impact on real software systems. Initial work by Gao et al. [10] found that static type checkers can detect a considerable percentage (15%) of public defects in *Javascript* projects. While prior work has laid the groundwork for the study of type-related defects in dynamically-typed languages, it is limited to the *Javascript* ecosystem. Other dynamically-typed languages may have different characteristics.

In recent years, there has been increased adoption of *Python* in open source projects [20]. This rise can partially be attributed to the explosive growth of machine learning, for which *Python*-based frameworks and libraries like Scikit-Learn, Numpy and Pandas have become common solutions. *Python*'s dynamic type system and a strong ecosystem of packaged modules empower developers to build functioning prototypes quickly. However, unlike *Javascript*, *Python*'s primary use case is not web development. In addition, *Python* does not have implicit type casting as in *Javascript*. Therefore, the nature of type-related defects in *Python* projects is likely to be different.

In this paper, we set out to quantify the potential benefits of applying a static type checker to *Python* projects. We operationalize one facet of those benefits by measuring the percentage of defects that were committed to a source code repository that could have been prevented if prior to the commit: (a) type annotations were present; and (b) the code was validated with a static type checker. Type annotations or type hints can be added to *Python* programs in a non-intrusive manner and do not affect the execution of *Python* programs. Static type checkers like `mypy` use the information provided by type annotations to perform static type checking. `Mypy` allows gradual typing of *Python* codebases; therefore, we can study their impact incrementally by adding annotations to selected regions of *Python* programs.

- *F.Khan, B.Chen and D.Varro are with the Department of Electrical and Computer Engineering, McGill University, Canada, E-mail: {faizan.khan3, boqi.chen}@mail.mcgill.ca, daniel.varro@mcgill.ca*
- *S.McIntosh is with the David R. Cheriton School of Computer Science, University of Waterloo, Canada, E-mail: shane.mcintosh@uwaterloo.ca*

We conduct an empirical study of all publicly available defects in 210 *Python* projects. For each defect in our sample, we revisit the version of the codebase just prior to the defect being fixed and add annotations. The annotated code is evaluated using a static type checker to determine if it can detect the defect. This data is then used to answer the following research questions:

RQ1 What proportion of Python defects could have been avoided by applying type checkers?
Motivation: By having a quantified measure of the software quality benefits of a static type checker, developers and project managers can get a clear idea of the improvements to expect when introducing a static type checker to a project's workflow. Therefore, we strive to quantify the quality benefits of using static type checkers by measuring the proportion of type-related defects that could have been prevented if a type checker like `mypy` was in use at the time of the commit.
Results: We find that 15% of the corrective defects (11% of all defects) in *Python* projects could likely have been avoided by using a static type checker. Our evaluation is conducted on fixed public defects. This number is likely to be higher for private defects observed during development, when the code has not gone through test cycles and external code reviews yet.

RQ2 What types of Python defects are being caught by type checkers? What types are being missed?
Motivation: Usage analytics are a powerful resource for improving software systems. However, such data is not available for type checkers like `mypy`. Without data to explain which type of defects are being caught/missed by static type checkers, impactful improvements to type checking tools are difficult to formulate. Therefore, we collect and analyze data about the type of defects that are caught/missed by a type checker.
Results: Incorrectly handled null objects are the commonly caught type-related defects, while defects requiring code additions/removals and value updates as fixes are regularly missed by static type checkers.

RQ3 What is the experience level of developers committing type-related defects?
Motivation: Understanding the experience level of developers committing type-related defects can help to better understand the users who stand to benefit most from the use of a static type checker. This will help tool developers to tailor their solutions towards a particular demographic. Thus, in this experiment, we set out to study the relationship between the experience level of developers and the occurrence of type-related defects.
Results: We find that there is no significant difference between the experience of developers committing type-related defects and the experience of developers committing non-type-related defects.

RQ4 Which code anti-patterns lead to type-related faults?
Motivation: Developers apply patterns and avoid anti-patterns when coding. When applied incorrectly, these patterns can cause faults during development. To avoid the faults caused by such patterns and to inform better adaptation of these patterns, an analysis is necessary. Therefore, in this experiment, we collect and analyze patterns of code that are the cause of recurring type-related faults.
Results: We find that incorrect handling of null values, dynamic attribute initializations and redefinition of *Python* references are a frequent cause of type-related faults. All of these anti-patterns affect the readability of large codebases and can be mitigated by the use of a static type checker.

The results of our study complement the findings of Gao et al. for *Javascript* projects [10] who reported that the use of a static type checker can catch between 11%-18% of defects in *Javascript* projects with a median of 15%. We find that for *Python* projects, static type checkers can catch 15% of the corrective defects (11% of all defects). Similarly, a deeper analysis into the types of defects caught/missed by `mypy` reveals that incorrectly handled null objects are commonly caught by type-checkers, whereas defects caused because of specification changes are commonly missed. Gao et al. reported similar findings for defects caught/missed by `TypeScript` and `Flow`.

Based on the findings above, and our extra analyses on type-related faults in *Python* projects, we recommend incorporating the use of a static type checker like `mypy` into the development workflow. Not only will it prevent type-related defects from being committed to the codebase, but also help in mitigating certain anti-patterns that can potentially lead to public defects.

2 Related Work

Static vs. dynamically-typed languages: Type systems have been extensively studied in the literature. We are not the first to compare the usefulness of static and dynamic type systems [6], [21]. These prior studies compare type systems along different dimensions: developer productivity, code usability and code quality.

Stuchlik and Hanenberg [37] conducted an empirical study on 21 subjects using *Java* and *Groovy* and found that small tasks were completed more quickly in *Groovy*. However, no difference was observed for larger tasks.

One of the largest analyses on the topic was conducted by Hanenberg [13], which included one year of data from 49 programmers. The programmers used either the statically or dynamically–typed variant of a language that was specifically designed for the study. The results indicated that static type systems did not have a significant positive impact on development time or code quality.

However, more recently Hanenberg et al. [14] observed that static types were beneficial to the maintainability of software systems. In particular, the statically-typed language used in the experiment outperformed its dynamically-typed counterpart in terms of understanding undocumented code and fixing type errors. However, no difference was observed when fixing semantic errors.

Moreover, Fischer et al. [8] compared the effect of type systems with respect to the effect of code completion by using *TypeScript* and *JavaScript* in Microsoft Visual Studio. The experiment measured the time taken to solve particular tasks, with language and the code-completion as two independent variables. The results showed a significant difference between *TypeScript* and *JavaScript*, with participants taking less time when using *TypeScript*.

Type-related defects : To the best of our knowledge, the most closely related work to ours is that of Gao et al. [10], who studied type-related defects in the context of Javascript. Our study builds upon and complements their work in four concrete ways. First, we study the Python ecosystem, which has different characteristics and developer culture than Javascript. Second, we analyze issue types using Swanson categorizations [38]. Third, we analyze the experience level of developers committing type-related defects. Finally, we analyze the anti-patterns that lead to type-related faults.

Partial type annotations : Bracha [2] proposed the concept of an optional pluggable type system that is independent of the language design. Such a type system, (a) has no effect on the run-time semantics of the programming language; and (b) does not mandate type annotations in the syntax. In such a setting, type-agnostic and type-aware tools can be used side-by-side.

Siek and Taha [35], [36] formalized the idea of combining strong and weak type systems by introducing the notion of a gradual type system. A gradual type system uses the partial information available about the known parts of types to detect inconsistencies among types. Typing conditions that cannot be determined statically are checked at runtime using casts. To achieve these goals, gradual typing extends an existing type system with an unknown type. In mypy and TypeScript , this unknown type is the default type of Any.

Type Inference : Gradual typing is syntactically similar to type inference [7], [18], [27]. Both approaches allow some type annotations to be omitted; however, type inference tries to infer the types of objects and rejects the program if it fails to do so. On the other hand, gradual type systems accept partial annotations and insert run-time casts for objects whose types cannot be determined statically.

Type Inference for Python : Several analyses have been developed to support type inference for Python. Salib et al. [34] inferred ow-insensitive types based on the Cartesian Product Algorithm (CPA). Cannon et al. [3] analyzed atomic types from the local view of procedures. Rigo et al. [30] and Gorbovitski et al. [12] optimized Python programs using abstract interpretation.

More recently, Hellendoorn et al. [15] used a deep learning model (Deeptyper) to infer types from an aligned corpus of Javascript and Typescript code snippets. Their results show that Deeptyper provided thousands of possible annotations (with over 95% accuracy) that were missed by the static type checker. An interesting avenue for future work would be to evaluate the results of the same model on a corpus of aligned data between Python code and type-annotated Python code and compare its performance with the model-based type-inferences discussed above.

Using GitHub as a data source : GitHub has been widely used as a data source for extracting insights about the bene ts of different features in programming languages on code quality. Ray et al. [29] conducted a study on a data set of 729 projects in 17 different languages, observing a signi cant relationship between code quality and the use of languages with strong and static typing.

However, other researchers offered a different perspective. Berger et al. [1] conducted a replication study of the Ray et al. paper and discovered factors that were unaccounted for in the data extraction and analysis phases. By controlling for factors such as sparse data and false-positive rates of defects, Berger et al. reduced the number of programming languages with signi cant results from 11 to 4.

In our study, we have been careful to control for factors pointed out by Berger et al., such as ltering for language-related defects and using a random sample for our analysis.

Listing 1. Code example with defect and the relevant x

```
1  class MongoRepository(object):
2      def byLocation(self, locationString):
3          record = self.db.provenance.find_one({'
               location':locationString})
4  ### fix patch
5  #++        if record is None:
6  #++            self.listener.unknownFile('id: '+str(uid))
7  #++            return
8          return self.inflate(record)
9
10     def inflate(self, record):
11         if 'duration' in record:
12             ...
```

3 Problem Definition

In this section, we present the scope of static type checking in the context of Python – a dynamically-typed programming language. First, an example is presented to illustrate the possible defects that static type checkers like mypy can catch. These defects are then presented in the context of the type annotations that are needed to catch such defects.

3.1 Motivating example

Listing 1 shows a snippet of Python code from the niprov project that tracks the provenance of brain imaging les. A fault is reported at line 11 because the function inflate expects a dict type as a parameter but instead a None object was passed. The function find_one returned a None object for the query and it was passed to inflate directly. The x, shown in the comments, handled this defect by reacting to None objects returned by find_one .

This snippet demonstrates a common problem in dynamically-typed languages. A developer must study the implementation or documentation of called functions to understand the types of accepted parameters and the types of returned objects. To an extent, developers must violate the principle of encapsulation to understand if their code will behave correctly.

Once type annotations are added to the code as shown in Listing 2, mypy can detect such problems without executing the code. Mypy will raise a warning for the function call at line 10, identifying the unexpected type of parameter. Thus, the defect could have been avoided if a type checker like mypy was in use at the point of the commit. Furthermore, as the example shows, the annotations serve as documentation within the codebase, thereby mitigating the need for looking into the external documentation or the de nition of a called function or API.

3.2 Type annotations in Python

Type annotations were added to Python in PEP-484 [41]. These type hints are not enforced by the Python runtime. However, once added, they are available in the abstract

Listing 2. Code with consistent and inconsistent annotations

```
1  ##stub example to be placed inside db.provenance
2  def find_one(param)->Union[None, dict]:  ...
3
4  class MongoRepository(object):
5      def byLocation(self, locationString):
6          record:Union[None, dict] = self.db.provenance.
               find_one({'location':locationString})
7
8          # inconsistent annotation
9          # record:dict = self.db.provenance.find_one
               ({'location':locationString})
10         return self.inflate(record)
11
12     def inflate(self, record:dict):
13         if 'duration' in record:
14             ...
```

syntax tree (AST) of the codebase and can be used by any tool that analyzes the codebase. Type checkers like mypy use the annotation information in AST to enforce type checking.

There are multiple locations within Python programs where one can add type annotations. Listings 1 and 2 provide examples of Python objects that can be annotated using type hints, i.e., function parameters, function return types and variable initializations. A complete list of the locations where type annotations may be applied can be found in the mypy documentation. [1]

3.3 Defects detectable by static type checkers

First, we define the type of defects that a static type checker like mypy can catch. We also formally define the type annotations needed to detect such type-related defects.

Let $D = \{d_1; d_2; \ldots; d_m\}$ be a set of defective programs together with their fixed versions $F = \{f_1; f_2; \ldots; f_m\}$ (assuming that $f_i = fix(d_i)$ for some function fix).

Type annotations: Let L be a dynamically-typed programming language like Python, and let L_a be a language based on L with support for type annotations. Let ann be an annotation function that maps a program $p \in L$ to $p_a \in L_a$, such that $p_a = ann(p)$. Then $mpc(p_a)$ is a type checking function that returns $\top$ (true) if an annotated program p_a is correctly typed using the mypy checker (i.e., no errors are indicated) and $\bot$ (false) otherwise.

The ann function may be an identity function in cases when no annotations are needed. In these cases, the program p is passed to mypy without any annotations ($p_a = p$).

Consistency of type annotations: While fault-triggering annotations can be added in many different ways, we only consider those that are consistent with the fix. Annotations are consistent if they trigger type-related faults when added to a defective version of the program; however, the same annotations, when used in the fixed version of the program $ann(fix(p))$, do not trigger any type-related faults.

The annotation in the comment at line 9 in Listing 2 shows an example of an inconsistent annotation. Adding a dict to record will trigger a type-related fault in both the defective version and the fixed version of the program. Therefore, this annotation is inconsistent with the fix and thus ignored. On the other hand, the rest of the annotations in Listing 2 are consistent, resulting in type-related faults only in the defective version of the program.

1. https://mypy.readthedocs.io/en/stable/getting_started.html

More formally, a type annotation ann is consistent when $mpc(p_a) = \top$ and $mpc(f_a) = \top$, where $p_a = ann(p)$ and $f_a = ann(fix(p))$.

This consistency filter is necessary to prevent us from adding ill-formed type annotations. It also serves as a stopping criterion for our annotation efforts. In practice, however, an annotation might trigger type faults in other regions of the fixed program f_a. This second-order effect may violate the consistency criterion for our annotations. However, we believe that these faults will not have happened if type annotations had already been in use by the project. Therefore, we strive to be consistent in our annotations as much as possible but do not fix type-related faults that are caused as a second-order consequence of our annotations.

Mypy-detectable defects: The total number of defects caught by a static type checker is then defined as follows:

$$D_{mpd} = \{d_i \in D \mid mpc(ann(d_i)) = \bot \wedge mpc(ann(fix(d_i))) = \top\} \quad (1)$$

Effectiveness of mypy: The effectiveness of mypy can be calculated as the percentage of defects detected by mypy over all fixed public defects (D_{pub_fixed} i.e. defects that are reported after the code is deployed to production, and have a valid fix attached with them.

$$D_{mpd\_fraction} = |D_{mpd}| \div |D_{pub\_fixed}| \quad (2)$$

D_{pub_fixed} is the set of all public defects that are fixed, and D_{mpd} contains public defects that are both fixed, and type-related Our experiment excludes public defects that are not fixed ($D_{pub\_not\_fixed}$), and private defects ($D_{private}$ i.e. defects observed by a single developer during local development before the code is deployed to production), which together form the set of all available defects in Python projects:

$$D_{total} = D_{pub\_fixed} + D_{pub\_not\_fixed} + D_{private} \quad (3)$$

While mypy can also detect private defects and public defects that are not fixed, we use public defects that are fixed (D_{pub_fixed}) in our experiments because they can be retrospectively validated in a concrete manner.

4 Experimental Setup

4.1 Corpus collection and filtering

Data source: We use GitHub Archive (GH-Archive) as our data source [19]. GH-Archive is a database of all GitHub activities, e.g. opening an issue, closing an issue or submitting a pull request. The entire database can be accessed via a public API or SQL queries.

Identifying Python projects: First, we collect the list of all Python projects available on GitHub. GH-Archive lists all GitHub repositories along with the programming language that contributes the most number of bytes to the project. We filter the repositories in GH-Archive for Python, which results in 551,366 projects that are predominantly implemented using Python [11].

Identifying defects in Python projects : Next, we collect the list of tracked defects in the Python projects. For this, we select all closed issues that are labelled as defects.

For practical reasons, we restrict our analysis to activity in the four-year period between 01-01-2015 and 12-31-2018. The starting point is imposed partly due to changes in the GitHub API starting from 01-01-2015, and partly because type hints were of cially added to Python in Sep-2014 (PEP-484) [41]. This ltering method results in 373,742 defects (recorded in the form of GitHub issue reports). Henceforth this collection of defects is referred to as the entire corpus[2].

Filtering the corpus for Python xes : The set of GitHub issues in our entire corpus includes xes to non-Python les as well. For instance, an issue in a Python project identi ed as a defect, can be a defect in the documentation with no changes required to Python les. Since these xes are out of scope for Python type checkers, we select the GitHub issues that (a) have an associated pull-request that, (b) contains changes to at least one Python le. This ltering criterion yields a ltered corpus of 10,916 Python issue reports, which spans 1,934 Python projects.

4.2 Sample collection

Our experimental procedure requires intensive manual analysis. Evaluation of the ltered corpus of 10,916 issue reports presents practical challenges. Thus, we select a sample of meaningful size, to which we apply our manual analysis.

In drawing a sample for analysis, following the guidelines of Krejcie and Morgan [22], we select a sample that is representative of the ltered corpus with a con dence level of 95% and a con dence interval of 5%. With a population of 10,916, a sample of 370 examples is needed to achieve the desired con dence level and interval. We round the sample size up to 400 for convenience, randomly selecting 400 issues from our ltered Python corpus.

Sample Statistics : Table 2 provides an overview of the size of projects in our sampled corpus (Table 1 provides the stars, sizes and contributors counts of the 210 projects). The minimum of zero in the Python row is from the EndlessSky-Discord-Bot project, where all Python les were removed as part of the x to the issue report in our sample. The table shows that the sample includes projects spanning a broad range of sizes from 332 to 5,735,321 LOC.

The Fix row shows the total number of modi cations (sum of lines added and removed) in the defect- xing commit. The sum of lines added and removed is an upper bound on the actual number of modi cations, since a single modi cation within a commit is shown as one line added and one line removed. As shown in the table, most defect- xing commits are small, with a median of 33 lines when compared to the median of 37,290 lines of Python code.

To con rm that the sample and population do not differ substantially in composition, we compare the distributions of issues in the top 10 projects in the sample and the ltered corpus. Table 3 shows these distributions sorted by the number of issues contributed. As can be seen in the table, the two distributions are similar, with eight out of ten top projects appearing at the same rank in both data sets.

2. The entire SQL query used for ltering is available at https://github.com/eff-kay/mypy-excursion

TABLE 1
The size (measured in LOC), stars and contributors for the 210 unique projects in our sample, checked out at the latest revision

	Min	Median	Mean	Max
Size	2	38,548	162,543	5,735,321
Stars	0	411	3,023	57,170
Contributors	1	57	216.18	5,846

TABLE 2
The size statistics (measured in LOC) of the projects and xes in our sample, checked out at the xed revision

	Min	Median	Mean	Max
Python	0	37,290	135,342	762,220
Total	332	95,533	250,551	5,735,321
Fix	0	33	609	47,140

TABLE 3
Projects ranked according to issue distributions. Columns 1 and 2 list projects from the corpus, columns 3 and 4 list projects from the sample.

	Filtered Corpus	No. of Issues	Sample	No. of Issues
1	Ansible	2,339	Ansible	101
2	Pandas	403	Pandas	13
3	Bokeh	325	Salt	11
4	Salt	212	Bokeh	9
5	Ansible-Modules-Core	210	Ansible-Modules-Core	6
6	Tribler	139	Micromasters	4
7	Astrophy	109	Tribler	4
8	Amy	91	Astrophy	3
9	Ansible-Modules-Extras	87	Amy	3
10	Cadaasta-Platform	77	Streamalert	3

Fig. 1. An overview of the steps in our mypy annotation procedure.

Small differences are detected in the tail because 11 projects contribute three issues to the sample. They are truncated here for presentation purposes.

Note that the sample of issue reports is only used in RQ1–RQ3. For RQ4, we analyze the codebases of the 210 unique projects within the sample, with each project checked out at the latest revision. More details about this difference in the artefacts can be found in Section 4.6.

4.3 RQ1: What proportion of Python defects could have been avoided by applying type checkers?

To measure the number of defects that could have been prevented by using a static type checker, type annotations are added to the defective regions of code until mypy can detect errors as shown in Figure 1.

Leveraging xes : Localizing the defective region in a program generally requires (re)execution of tests on a

Algorithm 1: Type annotation procedure

```
1  Input: D, The list of sampled defective code
2  Input: mpc, A function that checks whether the input source
          code type checks or not
3  Input: x , A function that applies the x patch to a defective
          codebase
4  Output: O, assessment of all sampled bugs
5  foreach d 2 D do
6      O:=[D.length];
7      Read the defect report and identify the PatchedRegion;
8      ann:= [] //A mapping that converts an input patch of
         code to code with annotations added;
9      foreach Identi er id in PatchedRegion do
10         if mpc(ann(d)) == ? ^ mpc(ann( x (d))) == > then
11             O[d] := True;
12             break;
13         end
14         else if author classi es d to be mypy-undetectable then
15             Note down the reason for failing to catch the
                 defect;
16             Categorize d into one of the undetectable types;
17             O[d]:= False;
18             break;
19         end
20         else
21             if type(id) == FunctionCall then
22                 if Function de ned in the same le then
23                     Annotate the function signature;
24                 end
25                 else
26                     Add a mypy stub for the function
                         signature;
27                 end
28             end
29             else
30                 Annotate the identi er;
31             end
32             update ann;
33         end
34     end
35 end
```

selected set of historical versions of a software system [4] [40]. In practice, it is dif cult to build and test each selected version, especially in large codebases (see Table 2). Therefore, we use contextual information from the xes to localize the defective regions. In Listing 1, the x patch in lines 4-7 implies that the defective code region is within the byLocation function. Contextual information like this is used when annotating the codebase.

4.3.1 Annotation tactics

Once the defective region is identi ed, the defective program is executed rst using mypy without any annotations. [3] By default, mypy assigns a default type of Any to objects whose type cannot be inferred from the code. For other objects, an inferred type is assigned at rst instantiation and then enforced throughout the rest of the program. In practice, this means that after the unannotated run, there will be errors that are not related to the particular x that is being investigated. To mask such unrelated errors, general type annotations are added.

3. To force mypy to type check the entirety of a function, at least one type annotation has to exist in the function declaration. Otherwise, mypy ignores every type check within the function. A typical annotation approach is to set the return type to None or Any.

Modules : The most common type of error in the unannotated mypy run is the failure to import modules or the failure to infer object types in external modules. To resolve these issues, mypy ags, such as ignore-missing-imports and follow-imports=skip, are used to set all module imports to Any. If the defective code depends upon objects in some external module, then mypy annotations are added in the relevant module and the module is passed explicitly as an argument in the mypy command.

If the relevant external module cannot be found locally, custom mypy stubs are created manually for the Python objects by inferring their type signatures from the online documentation, our cursory exploration of the object and its uses throughout the codebase. An online search query is formulated by (1) detecting the module from which the object is imported; and (2) combining the module's identi er and the object's identi er. Similarly, the local de nition of an object can be found with the help of any IDE or a tag traversal tool. We used VSCode's built-in commands to identify the local de nition of the objects. Furthermore, the uses of an object within a project can be found by using the search feature of VSCode. The object's identi er is used for the search query in this case.

Mypy stubs : Mypy stubs are typed interfaces for Python objects. These stubs are useful for creating interfaces for objects in modules that are dif cult to locate and modify locally. Since our focus is only on the defective region of code, the rest of the program can be viewed as a set of typed interfaces that the defective region of code depends upon. This allows us to minimize our refactoring efforts to reproduce a type-catchable defect.

Line 2 of Listing 2 provides an example of a mypy stub, where find_one is an external function with a return type of Union[None, dict] . Its typed signature is determined either from its documentation or implementation.

Annotation algorithm : Algorithm 1 presents our systematic annotation procedure. In summary, the de nition of mypy-detectability described in Equation 1 is used as a guiding principle for selecting the next possible annotation. A set of strict heuristics is used for categorizing defects that are not mypy-detectable (step 14). [4]

4.3.2 Determining public defects that are actually defects

Our collection of xed public defects relies upon issue reports being correctly labelled as defects by the authors. However, existing literature has shown that nearly 33% of the reports are incorrectly classi ed as defects [16]. Therefore, to get an accurate estimate of the total number of actual defects present in our sample, we classify the issue reports into three categories: Corrective, Perfective and Adaptive, following Swanson's classi cation [38]. Table 4 shows the extended Swanson categorization proposed by Hindle et al. [17]. To reduce the number of distinct categories, we classify the three new categories of New Requirements, Source Control Management and Code Cleanup, as Perfective changes.

Similarly, recent literature has classi ed defects into Intrinsic defects, i.e., defects that have a defect-inducing commit associated with them, and Extrinsic defects, i.e., defects that are caused because of external factors, such as

4. github.com/eff-kay/mypy-excursion/tree/appendix/heuristics/

TABLE 4
Extended Swanson categorization for classifying defects

Categories of Change	Issues Addressed
Corrective	Processing failure Performance failure Implementation failure
Adaptive	Change in data environment Change in processing environment
Perfective	Processing inef ciency Performance enhancement Maintainability New requirements Source control management Code cleanup

speci cation changes [32], [33]. However, the defects that are xed by Corrective changes appear to map onto Intrinsic defects, whereas the defects that are xed by Perfective and Adaptive changes are likely caused by Extrinsic factors.

In an ideal case, it would be preferable to extract a sample from a population of Intrinsic defects from the start. However, this information is not available in the GitHub corpus. Therefore, classi cation is carried out via manual analysis after an initial sample is collected.

Defect classi cation : To classify defects using the extended Swanson categorization, two authors independently conducted the classi cations. Then both authors compared their results, resolving the differences where an agreement could be reached. The heuristics used by both authors are shown in the online appendix. The nal categorizations resulted in a Cohen's Kappa score of 0.92, which indicates strong agreement. The remaining differences were resolved by tie-breaking votes from the other two authors.

We nd that the majority (219) of the xes in our sample (400) are Corrective, while a large number of the xes are Perfective (172) and a small number are Adaptive (9).

4.4 RQ2: What types of Python defects are being caught by type checkers? What types are being missed?

Defects that can and cannot be caught by mypy are investigated and classi ed into different categories. Open coding —– a qualitative research method for categorizing observations that lack a priori organization [26] —– is applied to identify emergent categories. The rst two authors independently coded all 400 of the issues. In a series of follow-up meetings, the codes were compared, merged, and re ned.

Caught defects: Caught defects are analyzed from the perspective of the type annotations required to catch the defect. A type-related defect is usually caused by an incorrect assumption about the possible type(s) that an object reference can hold. Therefore, the type annotations required to x a defect provides insight into the incorrect assumption that led to the defect in the rst place. Table 5 lists the possible types of defects caught by static type checkers.

Missed defects: For missed defects, we conduct the categorization by investigating the xes attached to the defects. The xes provide insight into the type of changes required to x the defect.

Since the xes show that the defect could have been caused by multiple defective regions of code, the classi cation is conducted at the change-set level. This means that one defect report can appear in multiple categories. Table 6 shows a sample of the categories of missed defects (the complete list can be found in the online appendix). Note that apart from New Tests and MissingDependency (see online appendix), all categories contain changes that are beyond the scope of static type checking.

TABLE 5
Categories of Caught Defects

	Category	Description
1	Value defects	Value defects are caused because of the value of an object being misused.
1.1	Optional-none	Optional-none defects occur when null objects are not handled properly.
1.2	False-vs-none	False-vs-none are incorrect logical conditions that occur because of the confusion between a boolean value of False and a null value.
2	Incorrect Type	These defects are incorrect assumptions about the type of an object.
2.1	General-incorrect-type	Defects in this category serve as a catch-all for all incorrect type defects not classi ed in other categories.
2.2	Byte-str-unicode	These defects are concerned with incorrect string assumptions, more common in Python2 where unicode characters and bytestrings are handled differently.
2.3	Incorrect-attribute	This category includes defects such as accessing a dictionary attribute that does not exist.
2.4	Incorrect-signature	Incorrect-signature defects happen because of the incorrect assumptions, either about the types of the parameters, or the return type of a function.
2.5	Int-expected	Int-expected defects occur due to the misuse of int types.

TABLE 6
Categories of Missed Defects

	Category	Description
1	Spec Changes	Speci cation changes include changes that modify the semantics of the code to conform to a certain speci cation. It also serves as a catch-all for issues not classi able other categories.
1.1	Branch Changes	Branch changes represent changes to the body of a control block.
1.2	New Class or Method	This category includes defects that require a new class or a method as a x.
2	Value Changes	Value changes represent changes that modify the values of an object rather than the semantics of the code.
3	Artefact Changes	These changes are made to artefact les that accompany the codebase.
3.1	New Tests	This category represents changes where entire new tests were added (either as new classes or new methods).

4.5 RQ3: What is the experience level of developers committing type-related defects?

In this experiment, we study the experience level of developers committing type-related defects for mature projects (with at least 1,000 commits). We measure experience ac-

cording to commit-based (artifact granularity) and review-based experience (task granularity).

Project commit experience : To calculate the project commit experience, we rst nd the commits at which the defective lines were added. We use the git-blamecommand for this purpose. This command traverses the project's history to the point at which a particular line was added to the codebase.Git-blameis executed for each type-defective line in the mypy-catchable defects in our sample (see Section 4.3). This provides us with a list of defective commits from which we extract the names of the authors who are implicated in the lines associated with type-related defects.

The project experience of each author is estimated by calculating the total number of commits by the author from the start of the project, to the point at which the type-related defect was introduced into the project. We use the git-log command for this purpose. This provides us with a list of defective commits and the corresponding author-experience values at the creation time of the defective commit. We lter out duplicate commits to get a list of unique entries.

As a baseline measure, we also calculate the experience of authors committing defects that are not type-related. We use the defective lines that are not part of the type-related defects to conduct the git-blamepart of the experiment. The rest of the procedure is the same as the one explained above.

File commit experience : In addition to calculating author experience at the project level, we also calculate the experience at a le level. The procedure is similar to calculating project-level experience except that we count the commits made to a defective le. We use the git-log command with a follow ag to retrieve commits to les.

Review experience: Thongtanunam et al. [39] showed that reviewing experience should not be overlooked when considering code ownership. Therefore, similar to Thongtanunam et al., we estimate the review-based experience of developers by counting the number of prior reviews that the implicated authors participated in.

More concretely, we start with the defective line associated with type-related defects. Then, we nd the defect-inducing commit and the defect-inducing author using the git-blamecommand. After that, using the GitHub API, we identify the pull-request to which the defect-inducing commit belongs to. Finally, we traverse backwards from the defect-inducing pull-request to the rst pull-request within the project, keeping a count of the number of pull-requests to which the defect-inducing author had served as a reviewer. This produces the reviewer experience of the author at the point of committing the defect-inducing commit.

We repeat the above process for every line, ltering out duplicated commits, since multiple defective lines can be part of the same defect-inducing commit. Similarly, as a baseline measure, we repeat the process for all defective lines that were not associated with type-related defects.

4.6 RQ4: Which coding anti-patterns lead to type-related faults?

In this experiment, we analyze the patterns of code that tend to be associated most with type-related faults. We evaluate the three projects that contribute the most issues in our sample, i.e., Ansible, Pandas and Bokeh (see Table 3). These projects vary in domain, age and size of development team.

To conduct the analysis, each project is checked out at the latest revision and evaluated against mypy. This provides us with a list of type-related faults for each le within the projects. A treemap representing the number of faults contributed by each le is constructed for all three projects. The top 20 les in each of the three treemaps are investigated for repeating code patterns that tend to co-occur with incidences of type-related faults.

Note that for this particular research question, we are not analyzing real-world defects, but local faults that a developer might observe while working with a dynamically-typed language. We also provide possible annotation strategies for these anti-patterns to help identify such faults.

5 Results

5.1 RQ1: What proportion of Python defects could have been avoided by applying type checkers?

We nd that 43 of the 400 studied defects are type-catchable using mypy. This is 10.75% of the total defects available in our corpus (calculated using Equation 2).

As reported in Section 4.3.2, there are 219 Corrective defects, 172 Perfective defects and 9 Adaptive defects in our sample. Since mypy was designed to detect type errors, which would require a Corrective x, the 181 Perfective and Adaptive xes are out of scope for our analysis.

Furthermore, our investigation of mypy-catchable defects leverages information in the xes to identify type-catchable defects. However, it is dif cult to correctly categorize defects just by looking at their code xes. Therefore, the initial investigation is conducted on all defects, regardless of whether it is a Corrective defect or not. This might result in some type-catchable defects that are not Corrective.

Therefore, after the initial investigation, we lter the Perfective and Adaptive defects out of the sample. This results in 33 defects that are both type-catchable and require a Corrective x. According to Equation 2, the total percentage of type-catchable defects is calculated to be $\frac{33}{219}$ = 15:06%.

Inter-rater experiment : In order to validate the ndings of this experiment, a second author independently reviewed a collection of 86 defects, which included the 43 mypy-catchable defects and 43 randomly selected non-mypy-catchable defects from the classi cations of the rst author. The analysis resulted in a Cohen's kappa score of 0.83 between the two authors, which indicates an “almost perfect” agreement [24]. In the rare cases where the authors disagreed, we conservatively downgraded the labels from mypy-catchable to non-mypy-catchable.

> Summary: 15% of the corrective defects (11% of all defects) in Python projects could likely have been avoided by using a static type checker.

Extra effort of annotation : To exploit the bene ts of a static type system, developers have to add annotations to their programs, which may increase development effort. Since Gao et al. [10] estimate this effort by measuring the extra tokens added to detect a type-catchable defect, we replicate this estimate for mypy.

The results of our annotation procedure show that on average, six annotations are needed to catch a type-catchable defect (with min, median and max values of 1, 5.04 and

TABLE 7
Categories of defects that were caught with mypy, and the total number of defects that belong to each category

Category	No. of defects
Optional-none	14
General-incorrect-type	8
Bytes-str-unicode	7
Incorrect-signature	4
Int-expected	3
Incorrect-attribute	3
False-vs-none	2

12, respectively). Unlike Gao et al. [10], we count the annotations added by mypy stubs as well. In addition, one annotation is added when mypy is imported, and a default return annotation for mypy to act on a particular function. In practice, these default annotations should already be available in a codebase that regularly uses mypy.

The annotations span 22 projects, with 95th percentile at 11 tokens, indicating that the number of tokens required to catch mypy-catchable defects will rarely exceed 11.

Similarly, a time-based analysis was conducted on another randomly selected sample of 100 defects spanning ten projects. The same annotation procedure as described in Figure 1 and Algorithm 1 was followed. A timer was started as soon as the issue report was opened, and stopped as soon as a conclusion was reached. The results show that on average 16.24 minutes are required to catch a type-related defect (with min, median and max values of 2.5, 15.12, 23.97 minutes, respectively).

However, none of these estimation techniques are able to faithfully measure the actual increase in development effort caused by adding type annotations for several reasons. First, the authors were guided by the knowledge of xes, whereas in a real-world setting, this knowledge would not exist. Second, a single token is counted as one unit of cognitive effort; however, different tokens can impose different cognitive loads for different developers. However, we believe that this limitation should be mitigated by the time calculation above. Third, the human subjects that conducted the annotations are not experts in mypy, and therefore might have used a greater number of annotations to catch a defect than a subject matter expert would use.

5.2 RQ2: What types of Python defects are being caught by type checkers? What types are being missed?

Table 7 groups the sampled defects based on the incorrect assumptions that led to the defect. The table shows that incorrect null handling (i.e., Optional-none) is the most commonly caught type-related defect. This is not surprising given that one must trace the entire life-cycle of an object to validate whether at any point its value becomes null or not.

The Gen-incorrect-type, which serves as a catch-all for all type-related defects, is the next most frequently caught defect. Furthermore, defects related to incorrect usage of text type are also frequently caught. We believe this is because in Python-2, text types, such as strings, unicodes, and bytes, are treated differently, which may confuse developers if the code is not documented properly.

TABLE 8
Top 5 undetectable defects with mypy, and the number of Corrective, Perfective and Adaptive defects in each category

Category of Missed defect	Corrective	Perfective	Adaptive
Spec Changes	108	123	6
Branch Changes	80	69	2
Value Changes	74	56	7
New Class or Method	32	58	2
New Tests	39	37	0

On the other hand, Table 8 shows the top ve types of defects that are missed by mypy. Unsurprisingly, Speci cation Changes are the most commonly missed changes among Corrective and Perfective defects. Speci cation Changes are hard to catch with mypy because they involve moving code blocks from one region to another thereby making it dif cult to add consistent annotations (see Equation 1).

Among the Adaptive defects, defects requiring Value Changes occur the most frequently. This is not surprising, since most Adaptive defects require xes that modify the code to handle changes in the external environment. In our sample, most of the Value Change defects are caused by changes to the structure of an external payload (APIs), the design of which is hard-coded in the defective codebase. Value Changes are dif cult to catch with static analyzers because parsers ignore the values of objects at compile time.

> Summary : Incorrectly handled null objects are commonly caught type-related defects. Defects requiring code block additions/removals and value updates as xes are regularly missed by static type checkers; however, such defects are beyond their scope.

5.3 RQ3: What is the experience level of developers committing type-related defects?

Figure 2 shows different beanplots comparing the experience counts of authors committing non type-related defects (upper curve) and the experience of authors committing defects that are type-related (lower curve) along different dimensions. All distributions are similar in shape (right skewed). Furthermore, Table 9 shows the quantitative comparison of the distributions using the Mann-Whitney U Test [23] (two-tailed, unpaired, = 0:05), and the Cliff's delta effect size measurement [5]. It can be seen from the table that the null-hypothesis (i.e. pairs of samples are drawn from the same distribution) cannot be rejected, for all types of experiences. Moreover, from the Cliff's delta column, we can summarize that the practical difference between the samples for all experience counts is negligible.

> Summary : There is no signi cant difference between the experience of developers committing type-related defects and those committing defects that are not type-related.

5.4 RQ4: Which coding anti-patterns lead to type-related faults?

This analysis resulted in the following anti-patterns that frequently lead to type-related faults.

TABLE 9
Cliffs Delta and Mann-Whitney U Test Results
(Null-hypothesis: Both samples are drawn from the same distribution)

Experience Type	Cliffs Delta	Mann-Whitney-U Test
Commits-Project-Level	0.080 neg	0.263
Commits-File-Level	0.039 neg	0.377
Review-Experience	- 0.034 neg	0.385

Fig. 2. Beanplot of author experience committing type-related defects and author experience committing non type-related defects. The dashed lines indicate the quartile points (25, 50, and 75 respectively)

5.4.1 Rede nition of Python objects

The rede nition of objects for different use cases throughout the life-cycle of a program is a common pattern in dynamically-typed languages. Although this exibility may reduce the development time for small prototypes, it can lead to readability issues when the project grows, or when multiple developers are involved in the project. Within the scope of the studied projects, we found three more speci c patterns of rede nition. We describe each pattern below.

"Poor man's" Strategy design pattern : The Strategy design pattern [9] allows designers to swap out algorithms (i.e. strategies) without the need to modify the client of the algorithm. This is achieved by specifying a base strategy interface, which each concrete algorithm must implement. Example: Listing 3 shows an example from Ansible (modules/system/hostname.py) where the Hostname class has strategy_class as a parameter. All subclasses that inherit from Hostname will provide a strategy_class parameter that implements an algorithm of choice as shown by RHELHostname. In this example, since the rst instance of the strategy_class has a type of UnimplementedStrategy , a static type checker will throw an error whenever strategy_class is rede ned in any of the sub-classes.

Conditionally de ned objects : In this anti-pattern, a reference is rst initialized with an object of type A and then conditionally reassigned to an object of type B . Example: Listing 4 shows an Ansible example where CustomHTTPSConnection is rst initialized as None, and then rede ned as a new class based on the attributes of httplib .

Rede nition within a try-catch block : Similar to the previous pattern, a type checker will raise a warning if an object is rede ned within a try-catch block.

Listing 3. Code extract: rede nition of objects in the "poor man's" Strategy design pattern

```
class Hostname(object);
  platform = 'Generic'
  distribution = None
  strategy_class = UnimplementedStrategy
  # Possible annotation strategy
  # distribution: Union[None, str] = None
  # strategy_class: Type[Strategy] = UnimplementedStrategy
class RHELHostname(Hostname):
    platform = 'Linux'
    distribution = 'Redhat'
    strategy_class = RedHatStrategy
```

Listing 4. Code extract: conditional rede nition of objects

```
CustomHTTPSConnection = None
if hasattr(httplib, 'HTTPSConnection') and hasattr(
        urllib_request, 'HTTPSHandler'):
    class CustomHTTPSConnection(httplib.HTTPSConnection):
        def __init__(self, *args, **kwargs): ...

###Possible annotation strategy
CustomHTTPSConnection:Union[None, Type[
        CustomHTTPSConnectionClass]] = None

if hasattr(httplib, 'HTTPSConnection') and hasattr(
        urllib_request, 'HTTPSHandler'):
    class CustomHTTPSConnectionClass(httplib.
            HTTPSConnection):
        def __init__(self, *args, **kwargs): ...

CustomHTTPSConnection = CustomHTTPSConnectionClass
```

Example: Listing 5 shows an Ansible example that preserves backwards compatibility with Python-2 by checking whether httplib is available or not. The type checker produces a warning because it explores both try and except blocks.

5.4.2 Initialization of sequence objects

Mypy requires sequence objects, lists or dictionaries to be explicitly typed by declaring the types of stored elements. Example: Listing 6 shows an example where mypy will report an error on line 2 because __path__ is not fully typed. The type inferred from the code is List , but mypy requires a full annotation that completely annotates the type of elements within __path__ .

5.4.3 Dynamic attributes

Assigning attributes dynamically to a Python class or a dictionary is dif cult for static type checkers to verify because complete type de nition is not available before execution. Example: Listing 6 shows an example from Ansible (module_utils/six/ __init__.py), where the attributes of _MovedItems are initialized dynamically based on the _moved_attributes list. A type error will be reported on line 14 because mypy does not recognize that moves has a dynamically assigned attribute builtins .

5.4.4 Null values

This is one of the most common anti-patterns of type-related defects (see Section 5.2). In this code pattern, a reference that may hold a None value is used in a code block that is not prepared to handle None values.
Example: Listing 7 shows a code extract where mypy will throw an error on line 7. The find_library function checks if an object with the given name exists, opens it, and returns the path; otherwise, None is returned. The typed interface of

Listing 5. Code extract: rede nition in a try-catch block

```
try:
    import httplib
except ImportError:
  # Python 3
    import http.client as httplib #type:ignore
    ### Fix to mitigate type errors
    ##import http.client as httpclienttemp
    ##httplib = httpclienttemp
```

Listing 6. Code extract: dynamic attribute initialization

```
class _MovedItems(_LazyModule):
    __path__ = []
    # annotation fix for 5.4.2
    #__path__:List[str] = []
    # annotation fix for 5.4.3
    #builtins:List[str]

_moved_attributes = [...]
for attr in _moved_attributes:
    setattr(_MovedItems, attr.name, attr)

_MovedItems._moved_attributes = _moved_attributes
moves = _MovedItems(__name__ + ".moves")
exec_ = getattr(moves.builtins, "exec")
```

Listing 7. Code extract: incorrectly handled None value

```
def find_library(name)->Union[str, None]:...
class CDLL(object):
    def __init__(self, name:str, mode=DEFAULT_MODE, handle=
        None, use_errno=False, use_last_error=False):...

libssl_name = ctypes.util.find_library('ssl')
# libssl_name: Union[str, None] = ctypes.util.find_library
    ('ssl')
libssl = ctypes.CDLL(libssl_name)
```

Listing 8. Code extract: Python mixin

```
class ExtensionOpsMixin:
    # Fix to mitigate type errors
    # @classmethod
    # def _create_arithmetic_method(cls, op):    ...
    @classmethod
    def _add_arithmetic_ops(cls):
        cls.__add__ = cls._create_arithmetic_method(
            operator.add)

class BaseMaskedArray(ExtensionArray, ExtensionOpsMixin):
    ...
class IntegerArray(BaseMaskedArray):
    @classmethod
    def _create_arithmetic_method(cls, op):    ...
```

the find_library can be seen at line 1. The CDLL class, on the other hand, does not expect a None value (see the typed interface of its __init__ function on line 3).

5.4.5 Python mixins

Mixins are a language concept that allows a programmer to inject code into a class to expand its functionality. Python mixins are units of functionality that are encapsulated in a class, which can be appended to other classes using multiple inheritance. Since mixins de ne methods that act on attributes that are not local, static type checkers will raise unidenti ed object errors on the usage of such attributes.

Example: Listing 8 shows an example of a Python mixin from the Pandas project, where ExtensionOpsMixin is a mixin that assigns a class method _create_arithmetic_method to the default add operation for the client class (lines 6–7). BaseMaskedArray then combines the mixin with the ExtensionArray type client class (lines 9–10). The _create_arithmetic_method is de ned by the relevant concrete classes that inherit from BaseMaskedArray , thereby creating a custom add operation for themselves.

Since create_arithmetic_method is not de ned within ExtensionOpsMixin , an error is reported.

5.4.6 Patterns that lead to defects in production

The patterns discussed above cause faults during development before the code is deployed. They may not lead to defects in production, which is the point of discussion in the previous RQs. To nd out if these patterns do lead to defects in production, we look for the patterns in the mypy-catchable defects in our sample studied in RQ1.

Table 10 presents the results of this analysis, which shows that incorrectly handled null values are a common cause of type-related defects in production, followed by the reference rede nition and dynamic attribute initializations. Type_mismatch serves as a catch-all for defects not categorized into any of the other patterns.

Summary: Incorrect handling of null values, dynamic attribute initializations and the rede nition of Python references are frequent causes of type-related faults. Reference rede nition is a bad practice in general, and affects the readability of a large codebase. Type checkers help to mitigate such code quality anti-patterns.

6 Practical Implications

Static type checkers can improve software quality : The results of RQ1 show that 11% of all defects (15% of the Corrective defects) could have been avoided if a type checker had been in use at the point of the commit. This result complements the ndings of Gao et al. [10] for Javascript projects, who showed that static type checkers can catch around 15% of the defects.

The 11-15% of defects caught by mypy is an underestimation of the improvements brought upon by type checkers. By enforcing type axioms during development, type checkers can reduce the number of unit tests required for validating types. Moreover, type checkers enforce type annotations, which can improve readability, and enable better code completion and navigation.

Correctly handling null objects can address a large proportion of type-related defects : As discussed in RQ2, incorrectly handled null objects account for a large proportion of the type-related defects caught using mypy (i.e. $\frac{14}{43}$ = 32:55%). A similar observation was made by Gao

TABLE 10
Occurrences of anti-patterns in the sample

Pattern	Counts
unmapped: type _mismatch	15
null values	12
rede nition	9
dynamic attributes	6

et al. when comparing TypeScript 2.0 (which employs null-checking) with TypeScript 1.8 (which does not). They observed an increase of 58% in the number of defects caught by TypeScript 2.0. This implies that just by correctly handling the null values, one can significantly reduce the number of type-related defects in dynamically-typed languages.

The confusion between bytes, string and unicode contributes to a large number of type-related defects in Python projects: As shown from Table 7, bytes-str-unicode is the third largest category among caught type-related defects. These defects are common in Python-2 codebases where the boundary between bytes and unicode is blurred. In Python-2, str is an abstract type that stores objects as bytes but can also be overloaded to store objects as unicode. Furthermore, an expression involving both bytes and unicode suffers from implicit conversion, which can lead to confusion about actual types. Python-3 avoids this confusion by storing str objects strictly as unicode and introducing a new bytes sequence class. This clear separation of boundaries catches the unicode-byte mismatch errors at static analysis time.

These defects are unique to Python projects and were not reported by Gao et al. for Javascript projects.

Static type checkers can mitigate anti-patterns during development: Table 10 lists the anti-patterns that can most commonly lead to type-related defects in Python projects. The table shows that in addition to the issues caused by incorrectly handled null objects, reference redefinition and dynamic attribute initialization can also lead to defects in production. Reference redefinition is a bad programming practice in general and affects the readability of the codebase, whereas dynamic attribute initializations can lead to increases in debugging time because the attributes are only available at runtime. The use of static type checkers will mitigate these anti-patterns during development.

7 Threats to Validity

Construct validity: Our analysis may detect a smaller number of type-catchable defects than in reality due to the strict criterion defined in Equation 1. Furthermore, since fixes are used to localize the defective regions, this limited annotation scope will omit defect-triggering annotations that are outside the scope identified by the fixes. This constraint is intentional as it serves as a good termination criterion on the time spent in annotating the code. Nonetheless, our observations should be interpreted as lower bounds rather than precise measurements.

Furthermore, leveraging fixes to localize the defective region implicitly assumes that developers will know about the defective region at the time of the commit. This assumption is practical, since most commits are likely to be small as shown in Table 2 (median of 33 lines).

Moreover, a variant of the SZZ algorithm [42] was used to determine the experience level of developers, which suffers from some known limitations [31]. These limitations were mitigated using a set of filters identified by McIntosh and Kamei [25]. More concretely, code comments and white space changes were not considered as part of the set of defective lines. In addition, defective lines committed after the date of original issue report were also filtered out. Finally, defective lines added as part of large change-sets (greater than 10,000 lines or 100 files) were also filtered out.

Internal validity: Our type annotation procedure from Section 4 was performed manually by authors who are not experts in mypy. If a defect was not detected using type annotations, then it was conservatively categorized as not mypy-detectable. A more precise type-annotation procedure carried out by a subject matter expert may yield more accurate results. Again, we encourage the reader to interpret our observations as a lower bound for the number of mypy catchable defects rather than an exact count.

External validity: To mitigate threats to external validity, we uniformly sampled issues from Python projects available on GitHub Archive. We do, however, restrict our analysis to the four-year period from 01-01-2015 to 12-31-2018, as explained in Section 4.1.

Our corpus is composed of fixed public defects, which is a subset of all Python defects. We use public defects because they are observable. We believe that this should not affect mypy-detectability, since despite our inability to study them, mypy can detect private defects as well.

8 Conclusion

In this paper, we study static type checking in the context of Python codebases. We find that 15% of the corrective defects in Python projects (11% of all defects) can be avoided by using a static type checker. In addition, we observe that incorrectly handled null objects and reuse of Python references are the main causes of type-related defects. Furthermore, we find that there is no significant difference between the experience of developers committing type-related defects and the experience of developers committing defects that are not type-related. Moreover, we provide a collection of anti-patterns in dynamically-typed languages that commonly lead to type-related faults, which to the best of our knowledge, have not been studied before.

All experimentation artefacts can be found at https://github.com/eff-kay/mypy-excursion.

References

[1] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek. On the Impact of Programming Languages on Code Quality: A Reproduction Study. ACM Trans. Program. Lang. Syst., 41(4), Oct. 2019.

[2] G. Bracha. Pluggable Type Systems. 2004.

[3] B. Cannon. Localized Type Inference of Atomic Types in Python. 06 2005.

[4] H. Cleve and A. Zeller. Locating Causes of Program Failures. In Proc. of the 27th Int. Conf. on Soft. Engg., ICSE '05, page 342–351, New York, NY, USA, 2005. ACM.

[5] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological Bulletin, 114(3):494–509.

[6] M. Daly, V. Sazawal, and J. Foster. Work in progress: an empirical study of static typing in ruby, 2009.

[7] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82, page 207–212, New York, NY, USA, 1982. ACM Press.

[8] L. Fischer and S. Hanenberg. An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability Using TypeScript and JavaScript in MS Visual Studio. In Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, page 154–167, New York, NY, USA, 2015. ACM.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.