How do third-party Python libraries use type annotations?

1nd Eric Asare
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
eric.asare@nyu.edu

2th Sarah Nadi
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
sarah.nadi@nyu.edu

Abstract

Type hints (a.k.a. type annotations) in Python have become an important tool for improving code readability, supporting static analysis, and enabling early bug detection.Accordingly, it makes sense for third-party libraries whose code is expected to be consumed by external clients to adopt type annotations for these benefits. However, it is unclear how pervasive the use of type annotations is in third-party libraries. In this paper, we investigate the extent to which third-party libraries have adopted type annotations and whether library developers place annotations where they offer the most benefit, such as on a library's public API. We analyze 76,327 versions across 11,021 libraries on the Python Package Index (PyPI). Our results show that 83.39% of libraries have at least one annotation. We identify six evolution patterns and find that most libraries maintain stable or gradually increasing annotation coverage, while only 4.13% abandon annotations once adopted. We find that library developers adopt type annotations to improve code clarity, linting, documentation, and static analysis, but may remove them when they cause errors or when the hints are considered unnecessary. Perhaps surprisingly, we find that developers prioritize annotating private functions over public/client-facing ones. Encouragingly though, many of these annotations are constrained and precise, with few instances of the Any type, most of which occur in parameters of public functions. Based on our findings, we provide implications for library developers and maintainers, package users, community contributors and researchers.

1 Introduction

Python remains one of the most popular programming languages [2, 51, 56]. Since Python is dynamically typed, developers do not need to declare the types of variables, functions, or other programming elements when defining them. However, the lack of type information has been shown to reduce the ability of type checkers to detect bugs in Python code [25, 58], as well as autocompletion support in Integrated Development Environments (IDEs).

Realizing the need for early detection of type-related bugs, Python introduced the notion of optional typing with the release of PEP 484 in May 2015 [57], continued with PEP 561 in 2017 [50], and most recently, PEP 749 in 2024 [62]. This became a middle ground that allows developers to optionally add type information (referred to as *type hints* or *type annotations*) to their code without completely changing the nature of the language. Specifically, these type hints are not checked or enforced by the Python interpreter. Instead, they enable static analysis tools such as Pyright [35] and MyPy [37] to detect type-related errors. After the introduction of type hints in Python, several empirical studies looked into the popularity and adoption of type hints. Most notably, in 2020, Rak-amnouykit et al. [49] reported that only 3.78% (2,678 out of 70,826) of Python repositories used type annotations. In 2022, Di Grazia and Pradel [7] found an increase to 6.92% (668 out of 9,655). The results of these studies suggest that the adoption of Python type hints increased over the years, although the overall adoption rates are low.

In addition to the data of these studies being at least three years old, both studies considered all Python projects on GitHub, without distinguishing the type of project. While it would be ideal to include type annotations in all types of projects, it seems that third-party libraries whose code is expected to be consumed by external clients would benefit the most. A recent study on the misuse of Application Programming Interfaces (APIs) of third-party Python libraries shows that type dynamics have a non-negligible impact on API usages in Python, where many incorrect assumptions about types are made during the invocation of a third-party APIs, leading to their incorrect behavior [19]. As one of the possible mitigation strategies, the authors suggest incorporating more type annotations to enable static checking or improving documentation. Industry developers also share the opinion that increased type annotations in third-party libraries is desired. For example, a recent discussion on the Python forum by an engineering manager[1] at Meta [34] suggests that industry developers also share the opinion that increased type annotations in third-party libraries is desired. The discussion shows that a third of the top 2,000 Python packages lack type annotations, leading to Meta and Quansight [48] collaborating to enhance type coverage for major libraries in the Python scientific stack (Pandas and NumPy); they have recently also invited the Python community to nominate additional libraries for this initiative [9, 59].

Compared to the low annotation usage found when considering any Python project [7, 49], the numbers in the above forum post suggest that popular Python packages may be higher adopters of type annotations. However, with more than 688,284 packages on PyPI, it is not clear whether the general Python package ecosystem is adopting type annotations as opposed to only the top packages. We also do not know how annotation usage in these libraries has evolved over time and how the decision to continue using or abandon annotations happens. Finally, given the nature and usage of third-party libraries, it would be useful to understand if the usage of type annotations differs between client-facing APIs (i.e., public APIs) versus privately used functions.

Accordingly, in this paper, we conduct a large-scale empirical study of the usage of type annotations in Python third-party libraries. We analyze type annotations on all functions from 76,327 versions across 11,021 libraries from the Python Package Index (PyPI) to answer the following research questions:

Conference'17, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Aaron Pollack is an engineering manager at Meta. Post url https://discuss.python.org/t/type-coverage-of-popular-python-packages-and-github-badge/63401

RQ1 What is the current adoption rate of type annotations in third-party libraries? With the aim of understanding the current annotation adoption in the Python package ecosystem, this RQ quantifies the extent to which third-party libraries use annotations in their latest release.

RQ2 What are the evolution patterns of annotation adoption? In RQ2, we focus on the subset of adopting libraries and study *how* they came about this adoption. For example, we want to understand if some libraries slowly grow their annotation coverage while others abandon annotations, as well as reasons behind such decisions. We use a mix of quantitative and qualitative methods in this RQ.

RQ3 Which functions do developers prioritise annotating? If developers prioritise the annotation of public functions, this maximizes external correctness, usability, and safety of the library's APIs. Prioritizing private functions on the other hand maximizes internal maintainability and refactoring safety. RQ3 explores which functions developers annotate, as well as the types they use in these annotations.

We find that third-party libraries are high adopters of type annotations, with 83.39% of the analyzed Python libraries using at least one type annotation. Among these adopting libraries, the median proportion of annotated functions (which we refer to as *annotation coverage*) is 19.68%. Thus, while the vast majority of libraries adopt type annotations, they generally have low adoption coverage.

We also identify six patterns of how developers annotate their packages over time. Notably, we find that 4.13% of libraries adopt annotations early and later abandon them, while 17.65% show steady growth and 3.87% perform gradual removal. The majority however (59.91%) remain stable with a constant coverage over time, which implies that they keep annotating newly added functions as well. Our qualitative analysis reveals that developers often adopt type annotations to enhance *linting*, *code clarity*, *documentation*, and to *fix static errors* reported by type checkers. From the issue conversations between library and client developers, we also find cases where annotations helped the library developers expose their blind spots, reveal bugs, and highlight potential incorrect API usage.

Perhaps surprisingly, we find that 63.35% of libraries prioritize annotating private functions over public ones, which is potentially concerning given for the client-facing API. Encouragingly though, most of the existing annotations are precise, with only a median of 2.33% of public function parameters annotated as `Any`.

Overall, our study reveals that while type annotation adoption is growing, coverage remains limited and uneven across libraries. Accordingly, client developers may need to rely on external stubs [47] or migrate toward libraries with more consistent annotation support to benefit from static analysis. Library developers should focus not only on adopting annotations but also on improving public API coverage to enhance usability and safety. Community contributors and initiatives such as those by Meta and Quansight can use our results to target low-coverage libraries and promote transparency through ecosystem-level incentives like annotation badges. Finally, researchers can build on this work to explore the relationship between annotation practices, API misuse, and software quality. To summarize, we make the following contributions:

```python
# 1. Untyped
def add(x, y):
    return x + y

# 2. Partially typed
def add(x: int, y):
    return x + y

# 3. Fully typed
def add(x: int, y: int) -> Any:
    return x + y
```

Listing 1: Gradual Typing in Python

- An empirical study that specifically focuses on annotation practices in third-party libraries. We study function-level type annotations in 11,021 Python libraries.
- A combination of quantitative and qualitative fundings that identify adoption trends and the reasons why some libraries continue to add or remove type annotations.
- An understanding of annotation prioritisation in libraries.
- Discussion of the implications of our results for client developers, community contributors, library developers, and researchers.

All our data and analysis code are available online on our artifact page: https://figshare.com/s/c6c4225c9dcb5cd3e507.

2 Background and Terminology

Dynamically Typed Languages. Dynamically typed languages such as Python, JavaScript, and Ruby perform type checking at runtime rather than at compile time. They allow variables to change type dynamically and do not require explicit type declarations. These languages are widely adopted for their flexibility and reduced development time [17, 42]. However, this flexibility also increases maintenance costs and makes programs more prone to run-time errors, particularly type errors [4, 11, 18, 39, 53].

Type Annotations and Gradual Typing. Type annotations, or type hints in Python, are a way to explicitly declare the expected types of program elements such as variables, function parameters, and return values. The Python interpreter does not enforce these type annotations so their use does not change how the code runs at runtime [57]. However, their presence allows static type checking of the code with third-party tools such as Pyright [35] and MyPy [37], which can catch type-related errors before runtime, improve code readability, and provide better support for IDE features such as autocomplete and refactoring. *Gradual typing* allows developers to progressively add type annotations to variables, functions and also symbols in a Python 3 codebase. Listing 1 illustrates the gradual addition of type hints to a function. In the first version of the `add` function, the function has no type hints or annotations, and is accordingly *untyped*. In the second version, an annotation is added to one of the parameters to indicate that it expects an integer, making the function *partially typed*. In the third version, both parameters are annotated as integers along with the return type, making the function *fully typed*. Note that the return type is annotated with `Any`, meaning the function can return any value. Static type checkers like MyPy or Pyright will not raise errors. This also applies to parameters annotated with `Any`.

```python
# Stub-based typing (PEP 561)
# File: add.pyi
def add(x: int, y: int) -> int:...

# File: add.py
def add(x, y):
    return x + y
```

Listing 2: Stub-based Typing

```python
def add(x: int, y: int) -> int:
    return x + y

result = add(5, "10")  # Type error: str cannot be
    added to int
```

Listing 3: Example of a Type Error in Python

Typing Mechanisms: Inline vs Stub Files. Python supports two main mechanisms for providing type annotations: Inline annotations (PEP 484) [57] are written directly in the `.py` source code (e.g., the type annotations in Listing 1) while using Stub files per PEP 561 [50] keep type information in separate `.pyi` files. Listing 2 shows how we can fully type the add function that exists in `add.py` in a separate corresponding stub file `add.pyi`. Stub files may be included directly in a package or distributed separately on PyPI [46] e.g., `pandas-stubs`[2]. Stub files can also be contributed to *Typeshed* [47], a community effort to contribute type hints for the Python standard library and some third-party libraries. Our study focuses on inline and stub annotations that developers have added *within* their own packages.

Static Type Checkers. Several static type checkers have been developed to leverage Python's optional typing system. Notable examples include MyPy [37], Pytype [12] (Google), Pyre and Pyrefly [10, 32] (Meta), Pyright [35] (Microsoft), and Pyinder [40]. These tools analyze annotated code to detect type inconsistencies before execution. Listing 3 shows an example of a type mismatch that such detecters can catch before the code is executed. This type error typically appears in the client's IDE if the type checker is integrated, or the client can run the type checker on the Python file. For example, `mypy add.py` to see the same error.

Private vs Public Functions. While Python does not have explicit access modifiers (i.e., `private` vs. `public`) to separate client-facing APIs from internal functions, various conventions can indicate whether a function/variable is intended for internal or external use:

- Conventionally, a leading single underscore (`_myfunc`) marks a function as *private* while no leading underscore signals that it is a *public* API.
- However, a module-level list `__all__` can explicitly define which symbols form the public API, even if they start with an underscore.

Listing 4 illustrates these conventions. We refer to each of these functions when annotated as *private annotated* or *public annotated* in our analysis.

Pyright's Type Completeness Feature. Pyright is a widely adopted static Python type checker. It provides a unique *type completeness*[3] feature that displays and quantifies how fully typed a project's public interface is. Pyright defines a codebase as *type complete* if all public symbols in its interface are annotated with fully known types. Private symbols are excluded from this calculation. A function or method is considered *type known* if: (1) All input parameters have annotations, (2) The return type is annotated and known, and (3) The application of decorators preserves a known type. Essentially, Pyright will mark a function that is not private as type known if it is fully typed as shown in the third example of Listing 1.

[2]https://pypi.org/project/pandas-stubs/
[3]https://microsoft.github.io/pyright/#/typed-libraries?id=library-interface

```python
# Module: example_module.py

__all__ = ['public_func', 'also_public']

def public_func(a: int, b: int) -> int:
    """Public annotated function: listed in __all__ and
        has type annotations."""
    return a + b

def also_public(x):
    """Also public but unannotated"""
    return x * 2

def _private_func(y:str) -> str:
    """Private annotated function: single underscore
        prefix and annotated"""
    return y - 1

def _helper():
    """Private helper function and not annotated"""
    pass
```

Listing 4: Example of Private vs Public Functions

```
{
    "category": "function",
    "name": "matplotlib.pyplot.plot",
    "referenceCount": 1,
    "isExported": true,
    "isTypeKnown": true,
    "isTypeAmbiguous": false,
    "diagnostics": []
}
```

Listing 5: Pyright JSON output for a function in matplotlib

To assess a library's type completeness, Pyright can be run with the following command:

```
pyright --outputjson --ignoreexternal --verifytypes <lib>
```

First, this checks for the presence of a `py.typed` file, as required by PEP 561 [50], which mandates that library developers and package maintainers include this file to declare that their library is typed. If this file is not found, Pyright cannot run successfully. Once the file is found, Pyright analyzes the library files, extracts the annotation information, and computes the type completeness score for all programming elements that form the interface, including type aliases, variables, constants, and functions. For example, running the Pyright command on `matplotlib` returns a completeness score of 0.54. This completeness score applies to all public interface symbols. In our study, we focus only on functions. Listing 5 shows the function-level details available in the output, such as symbol names, export status, and type knowledge for fine-grained analysis. We discuss how we use Pyright to perform our analysis in Section 4.2.

Third Party Libraries. Third-party libraries are integral to modern software development. They provide ready to use code for many different functionalities supporting software developers [38]. Open-source software projects heavily rely on third-party libraries [61]. In the Python ecosystem, libraries (or packages) can be downloaded from PyPI. As of October 2025, PyPI contains 688,284 libraries/packages, 7,496,956 releases and 965,056 unique downloaders [46].

3 Related Work

We discuss the existing research studying Python type annotations, and how it motivates our work.

Type Annotation Adoption and Practices. Rak-amnouykit et al. [49] report low adoption rates of Python 3-style annotations, with only 2,678 out of more than 70,000 repositories (3.78%) containing partial annotations as of March 2019. They find that developers primarily annotate function parameters, followed by return types, and less frequently global assignments. They also find that developer-written annotations are often difficult for type checkers like PyType or MyPy to validate, with only 15% of typed repositories being type-correct in MyPy. Two years later, DiGrazia and Pradel [7] analyzed 1.4 million annotation changes across 9,655 GitHub projects created between 2010 and 2019 and found that type annotations are increasingly popular but still uncommon: only 7% of projects used type annotations at all in 2021, and even in annotated projects, less than 10% of possible code elements are annotated. They found that developers prefer annotating function arguments and return types. By analyzing the evolution of the type annotations at the commit level, they identify three evolution patterns: regular annotation, type sprints, and occasional use. While they do not delve into the reasons different projects adopt/abandon annotations, they found that projects with more contributors tend toward regular annotation, while smaller teams follow occasional use. They also found a positive correlation (Pearson coefficient 0.704) between the number of annotations and detected type errors using Pyre [10], suggesting that annotations help reveal errors. Both studies analyzed all available GitHub repositories, without distinguishing between third-party libraries, system projects, or personal/toy projects.

Reflecting on the above work, we see that the adoption of type annotations increased between 2019 and 2021, but is still low overall. While having type hints for any Python project is desirable, we believe that it is even more important for third-party libraries that offer APIs that other clients depend on. For example, recent work by He et al. [19] shows that type dynamics and typing problems are the main cause of API misuse of Python libraries. Furthermore, recent discussions on Python forums indicate that about one-third of the top 2000 PyPI packages have no type annotations, while the remaining 70% have some coverage or Typeshed stubs[4]. Such discussions suggest that the developer community understands the value of type annotations in third-party libraries. Inspired by this, our work studies a broader set of Python packages, not limited only to popular ones, to understand ecosystem-wide annotation rates and evolution patterns in third-party libraries, using a combination of quantitative and qualitative analyses. Additionally, we dive into understanding annotation adoption at the API level, distinguishing between client-facing APIs and private functions.

Type Annotations and Defects. Chen et al. [4] conduct an empirical study on nine large Python systems (over 460K LOC) and find that inconsistent variable types are common and significantly correlate with bugs, emphasizing the risks of dynamic typing. Lin et al. [28] study type annotation-related defects across 13 Python projects, proposing a taxonomy for defect classification. Xu et al. [58] investigate three type checkers (MyPy, Pyright, PyType) and find that adding type annotations significantly improves bug detection, observing that 29 out of 40 bugs were detected after annotation compared to 14 before. This finding, which highlights the critical role of function parameter annotations in bug discovery, reinforces the importance of studying annotation patterns at the function level, particularly for client-facing API functions in libraries. Similarly, Khan et al. [25] report that 15% of defects in their dataset could have been detected and prevented by MyPy if type annotations were added. We leverage the Pyright static type checker in our analysis to identify annotated functions. However, instead of focusing on defect detection and the efficacy of specific type checkers, our goal is to understand the broader evolution and adoption of type annotations in Python third-party libraries.

Automatic Generation and Fixing of Type Annotations. Research in this category focuses on techniques to automatically infer and generate type annotations or fix type errors in Python projects. Guo et al. [16] analyzed existing inference tools, noting their difficulty in achieving high coverage and accuracy simultaneously, and implemented a tool to demonstrate pathways for improvement. Sun et al. [54] presented Stray, a static type recommendation approach designed to improve type correctness by leveraging program analysis instead of machine learning. Furthermore, Chow et al. [5] developed PyTy, the first learning-based repair technique for fixing Python type errors, demonstrating its real-world efficacy with 20 out of 30 suggested fixes being merged by developers. We do not explore generating type annotations in our work.

Summary. Overall, we find that most existing studies focus on file- or project-level changes and usually consider any Github Python repo/project regardless of its nature. Our work complements the ongoing body of work to understand type annotations in the Python ecosystem. Specifically, in contrast to prior studies, we (1) focus on third-party libraries that are expected to be used by down-stream clients and (2) investigate a granular, function-level perspective to understand how type annotations are adopted and prioritized across the wider ecosystem of third-party libraries.

4 Empirical Study Setup

Figure 1 shows an overview of the data we collect and analyze to answer our three research questions. In this section, we discuss the top part of that figure, which represents the common data collection and annotation extraction steps. We explain the metrics and additional methods used to answer each RQ in their respective sections (Sections 5-7).

[4]https://discuss.python.org/t/type-coverage-of-popular-python-packages-and-github-badge/63401

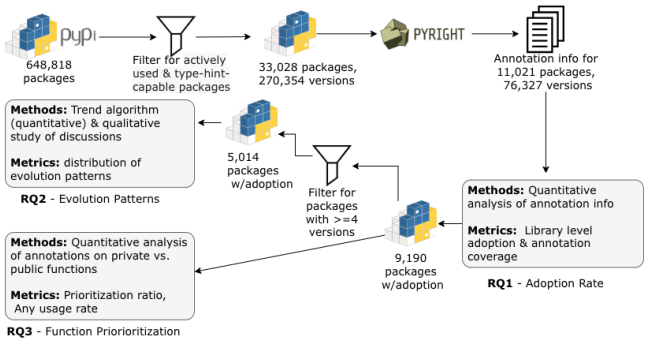

Figure 1: Overview of our empirical study

4.1 Studied Libraries

To collect all available Python third-party libraries, we query the public PyPI dataset [8] on Google BigQuery [13] for all packages downloaded in May 2025. This provides us with 648,818 packages. To focus on actively used and type-hint-capable packages, we apply several filters.

- We only retain packages with more than 100 downloads to focus on actively used libraries. The threshold is based on our frequency distribution analysis, which showed that packages with <100 downloads form a long tail and rarely represent actively used libraries. This step filters out 286,641(44.18%) packages.
- We exclude 3,150 (0.49%) packages that contain only stub files in order to focus on libraries that offer APIs that can be used by other client code.
- We include only libraries whose latest release supports Python 3 (required for type annotations) and that have at least one major or minor release. This criterion filters out 144,344 (22.25%) packages.
- We only retain packages marked as *Production* or *Stable* in their metadata to focus on well-tested, reliable libraries rather than experimental ones. This filters out 181,655 (29.99%) packages.

We extract the annotation information in the remaining 33,028 (5.09%) packages. The total number of versions corresponding to these packages is 270,354.

4.2 Annotation Extraction

As a prerequisite to answering all our RQs, we need to analyze each library's source code to identify type annotations in its functions. We focus primarily on functions in this study, because third-party libraries typically expose their functions for client use and static analysis. We use Pyright [35] to identify annotated functions. As discussed in Section 2, Pyright's default behavior is to return type completeness information only for public interface symbols. We modify Pyright to return type completeness information for both public and private symbols. We also modify its output to include each function's parameters and their types, the return type, and the full function signature.

To analyze different library versions with Pyright, we proceed as follows. (1) we install the target library version in an isolated virtual environment using `pip` and record the location of its installed source code. (2) If the package's source code does not already include a `py.typed` marker file, we add an empty one at the package root to enable Pyright's type analysis. (3) We then run our modified version of Pyright locally with the `–verifytypes` flag to extract symbol information. (4) For each successfully analyzed version, we store the extracted symbol information in the corresponding library output file. Finally, (5) we retrieve the release and creation dates for all successfully analyzed versions from PyPI.

We parallelize the process of collecting the symbol information for the 270,354 versions of the 33,028 packages on our university's High Performance Computing (HPC) platform. To enable replication of our analysis on similar HPC platforms, our online artifact: https://figshare.com/s/c6c4225c9dcb5cd3e507 includes a Singularity environment that contains the Python version, the modified Pyright and Node versions, the analysis results (including failed cases), the HPC job submission files, a setup README, and all data collection scripts. Our data collection ran across 3,500 CPU cores for approximately 72 hours.

Getting the symbol information for 194,027 package versions across 22,007 libraries failed. Upon investigation, the reasons include missing packages i.e pip cannot find the package (likely due to renaming), missing versions (probably yanked) on PyPI, problems with the package's pre-built wheel distribution (i.e., a broken wheel), or Pyright failing to detect the `py.typed` file. In some cases, even when `py.typed` is present, Pyright cannot detect it because certain libraries structure their projects differently, so programmatically adding a `py.typed` file to the package entry does not trigger Pyright's detection process.

Overall, we collect a complete set of labelled symbols per version for each of 76,327 versions across 11,021 third-party Python libraries. This is the final data we use to answer our research questions.

Table 1: Current adoption and coverage of type annotations in third-party libraries (latest release).

Metric	Value
Libraries with Adoptions	9,190 (83.39%)
Median Coverage (Adopting)	19.68%
Mean Coverage (Adopting)	34.55%
Libraries without Adoptions	1,831 (16.61%)
Total Libraries	11,021 (100%)

5 RQ1: What is the current adoption rate of type annotations in third-party libraries?

Methods. We consider a function as annotated only if *all* its arguments and return type are annotated, i.e., it is marked by Pyright as `istypeknown`. This ensures consistency with Pyright's notion of completeness and avoids inflating coverage through partially annotated functions that do not provide full type information to users or type checkers. Based on this, we define the following metrics to study the adoption of type annotations.

- *Library-level adoption*: A library is considered to have adopted type annotations (i.e., is an *adopting library*) if at least one of its functions in the latest release is annotated.
- *Annotation coverage*: For each adopting library, we measure the percentage of functions that are annotated.

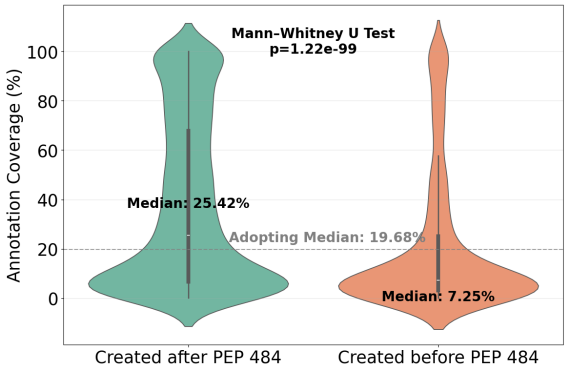

Figure 2: Distribution of annotation coverage across 9,190 adopting libraries, separated by creation date.

Results. Table 1 shows that out of the 11,021 libraries we study, 9,190 (83.39%) contain at least one function with type annotations, while 1,831 (16.61%) do not have annotations for any of their functions. This indicates that type annotations are present in a substantial subset of the ecosystem.

We now focus on the 9,190 libraries that have adopted type annotations to understand the variance in annotation coverage. We summarize the distribution of annotation coverage across all adopting libraries, with the median and mean values in Table 1. We find that the median annotation coverage is 19.68%, meaning that in a typical adopting library, only about one fifth of the functions are annotated. The mean coverage is 34.55%, indicating that a small number of highly annotated libraries pull the average upward. The violin plot in Figure 2 shows that a large concentration of the libraries (regardless of their creation date) are around the 10% coverage range and another much smaller concentration close to the 100% coverage. More precisely, we find that 3,471 (37.8%) of libraries fall within the 0–10% coverage range, while surprisingly, around 1,029 (11.2%) have coverage between 90% and 100%.

Table 2 shows a sample of five libraries from each of the two extreme coverage ranges, along with their descriptions and popularity and size characteristics. We can see that some very popular libraries (e.g., scipy) have low annotation coverage while less popular libraries (e.g., humanize) have high annotation coverage. We run a Spearman correlation test to investigate if any of these characteristics generally correlate with annotation coverage. However, since not all packages on PyPI have corresponding GitHub repositories, we could run this correlation analysis for only 6,691 libraries. We find that larger projects (by LOC) tend to have slightly lower annotation coverage (Spearman $r = -1.75e-01$, $p < 2.64e-49$). Popular projects (by stars) also show a slight negative correlation with annotation coverage ($r = -1.33e-01$, $p < 9.05e-29$) as do projects with more contributors ($r = -1.41e-01$, $p < 2.35e-32$). We speculate that since large projects have more code to annotate, high coverage becomes harder to achieve. It could also be the case that annotation adoption decisions are made even before a library becomes popular by stars or amasses a large contributor base. We do not find statistically signficant correlations between annotation coverage and number of downloads suggesting that current activity levels are not the main driver of adoption.

Since type hints became available in Python 3.5 in September 2015 [57], libraries created before this release may not have adopted them, whereas libraries created afterward are more likely to do so. Accordingly, we compare the current adoption rates of libraries created before September 2015 (legacy) and those created after (modern). Our dataset contains 1,743 legacy libraries and 7,442 modern libraries. The violin plot in Figure 2 shows that modern libraries exhibit a higher median annotation coverage of 25.42%, compared to 7.25% for legacy libraries. A Mann–Whitney U test confirms that this difference is statistically significant (p = 1.22e-99), suggesting that older projects were slower to adopt type hints, whereas newer libraries gradually embrace PEP 484. We further study evolution patterns in RQ2.

Discussion. Together, these findings confirm findings from prior studies that type annotations exist in Python repositories [49] and are becoming increasingly popular [7]. It is interesting to note that even though we analyze 14.4% more projects than the most recent study in 2022 [7], we observe a much higher adoption rate than the previously reported 7%. Our results suggest that in addition to a potential overall growth in annotation adoption, third-party libraries are more likely to adopt annotations than less "client-facing" code. Among the adopting libraries, a large proportion (37.8%) have at most 10% of their functions annotated while only 11.2% have at least 90% of their functions annotated.

RQ1 Summary: 83.39% of Python libraries use type annotations. However, the median annotation coverage across adopting libraries is only 19.68%, with (37.8%) have at most 10% annotation coverage while only 11.2% have at least 90% annotation coverage.

6 RQ2: What are the evolution patterns of annotation adoption?

Methods. To answer RQ2, we focus on the subset of adopting libraries and use a mix of quantitative and qualitative methods. We consider adopting libraries with at least 4 releases to have enough data points to establish meaningful trends. We only consider major or minor releases, and exclude patch-only releases. In total we consider 5,014 adopting libraries to identify evolution trends. For the quantitative trend analysis, we use the Mann-Kendall (MK) test [22] to detect monotonic trends in adoption. The MK test returns an increasing/decreasing output if a statistically significant growth/decrease is detected; otherwise, it returns no trend.

We use Algorithm 1 to categorize each library's adoption into one of six trends. For each library, we sort its releases (R) chronologically ($R = [R_1, \ldots, R_n]$) and consider the annotation coverages (A) of each release $A = [Ann_1, \ldots, Ann_n]$. We compute the maximum adoption coverage *max* across all releases (Line 1). If the annotation coverage of the latest release of the library has dropped below 20% of the max adoption found in this library (i.e., dropped by more than 80%), we consider that this library has *Adopted and Abandoned* type annotations (Lines 2-3). If neither of these cases happen, then we start analyzing data trends. We first calculate the standard deviation of coverages σ on Line 5. If the standard deviation is within 5%, then we consider the evolution of this library to be *Constant*, i.e., no growth or drop (Lines 8-9). This initial criteria allows a "buffer" of deviations that may incorrectly cause the trend analysis to detect decreasing/increasing trends for small changes that are not always

Table 2: Sample of libraries with annotation coverage and characteristics.

Library	Coverage	Description	Stars	Contrib.	Downloads	LOC
MXNet	0.98%	Deep learning framework designed for both efficiency and flexibility	20.8K	866	778.7K	119.2K
XBlock	4.91%	Open edX component architecture for building courseware	467	94	98.5K	6.4K
svgelements	4.99%	High fidelity SVG parsing and geometric rendering	162	10	175.9K	12.5K
fastai	8.34%	Simplifies training fast and accurate neural nets using modern best practices	27.5K	247	652.8K	14.9K
scipy	9.42%	Fundamental algorithms for scientific computing in Python	14.1K	1.6K	20.5M	292.2K
Flask	91.64%	Simple framework for building complex web applications	70.5K	732	142.4M	10.3K
pytest	92.31%	Simple powerful testing with Python	13.2K	915	244.2M	61.2K
scalene	94.34%	High-performance CPU, GPU and memory profiler for Python	13.0K	57	238.1K	14.4K
Blacksmith	97.49%	REST API Client designed for microservices	9	4	859	10.4K
humanize	100.00%	Convert data into human-readable formats	662	37	26.8M	2.0K

meaningful. Otherwise, we consider MK_{trend} (Line 6) as well as the recent slope s (Line 7) to categorize the evolution. We define the *recent* slope as the slope of a linear regression fitted to the last 25% of data points, capturing the trend direction of the most recent values. Note however that if fewer than two data points fall within this recent window, s cannot be reliably estimated and we set to zero, as it would not provide meaningful information beyond MK_{trend}. Lines 10 onwards show how we use the combination of these two values to determine trends. If the MK_{trend} is increasing but the recent slope is strongly negative (<-1), then we categorize the trend as *Grew and Dropped* (Lines 10-11). Conversely, If the MK_{trend} is decreasing but the recent slope is strongly positive (>1), then we categorize the trend as *Removed and Rebounded* (Lines 12-13). If MK_{trend} is increasing and the recent slope is also positive (>0), we consider it as *Grew over time* (Lines 14-15). Based on our examination of many evolution graphs, we also consider the case where the MK_{trend} failed to detect an overall trend but there's a strong positive recent slope (>1) as *Grew over time* (Line 14). The inverse applies for *Removed over time* (Lines 16-17). Finally, if none of the above patterns are detected, we categorize the library's adoption as *No Pattern* (Line 19).

To further understand the adoption trends observed in our quantitative analysis, we conduct a qualitative analysis on a statistically representative sample of 68 libraries (90% confidence level with a 10% margin of error). To ensure adequate representation from each evolution category, we apply proportionate stratified sampling [6, 27]. We round up fractional sample sizes to the nearest whole number (e.g., 3.3 becomes 4), which results in a total of 71 libraries: *grew_over_time* ($n = 13$), *adopted_and_abandoned* ($n = 3$), *grew_and_dropped* ($n = 1$), *removed_over_time* ($n = 3$), *removed_and_rebounded* ($n = 1$), *constant* ($n = 41$), and *no_pattern* ($n = 9$). For each selected library, we examine its evolution graph to identify notable changes and relate these to GitHub issues or developer discussions that may provide insights into the motivations behind adopting or abandoning type annotations.

Quantitative Results. We run Algorithm 1 on 5,014 adopting libraries with at least four releases. Table 3 shows the number of libraries in each evolution pattern. We now discuss each pattern

Algorithm 1 Evolution Trend Analysis

Require: Annotation percentages $A = [Ann_1, Ann_2, \ldots, Ann_n]$
Ensure: Classification of trend of Library L

1: $max \leftarrow \text{maxAdoption}(A)$
2: **if** $Ann_n < 0.2 \times max$ **then**
3: **return** *Adopted and Abandoned*
4: **else**
5: $\sigma \leftarrow \text{std}(A)$
6: $MK\_trend \leftarrow$ Mann–Kendall(A)
7: $s \leftarrow \text{linearRegressionSlope}(A[\text{last } 25\%])$
8: **if** $\sigma \leq 5$ **then**
9: **return** *Constant*
10: **else if** (MK_trend = increasing **and** $s < -1$) **then**
11: **return** *Grew and Dropped*
12: **else if** (MK_trend = decreasing **and** $s > 1$) **then**
13: **return** *Removed and Rebounded*
14: **else if** (MK_trend = increasing **and** $s \geq 0$ **or** MK_trend = no trend **and** $s > 1$) **then**
15: **return** *Grew Over Time*
16: **else if** (MK_trend = decreasing **and** $s \leq 0$ **or** MK_trend = no trend **and** $s < -1$) **then**
17: **return** *Removed Over Time*
18: **else**
19: **return** *No Pattern*
20: **end if**
21: **end if**

presented in the Table, while mentioning an illustrative example from that pattern as shown in Figure 3.

We find that more than half of the libraries (59.91%) had relatively constant adoption across their examined releases (e.g., pytest [45]). A small proportion of 4.13% libraries adopted annotations but abandoned more than 80% of their maximum coverage by the latest release (e.g., fastai [21]). Conversely, a larger proportion of 17.65% libraries' adoption grew over time (e.g., flask [41]), although a small fraction (0.90%) initially grew but dropped coverage in their latest release (e.g., emoji [26]). Over time, 3.87% of libraries consistently

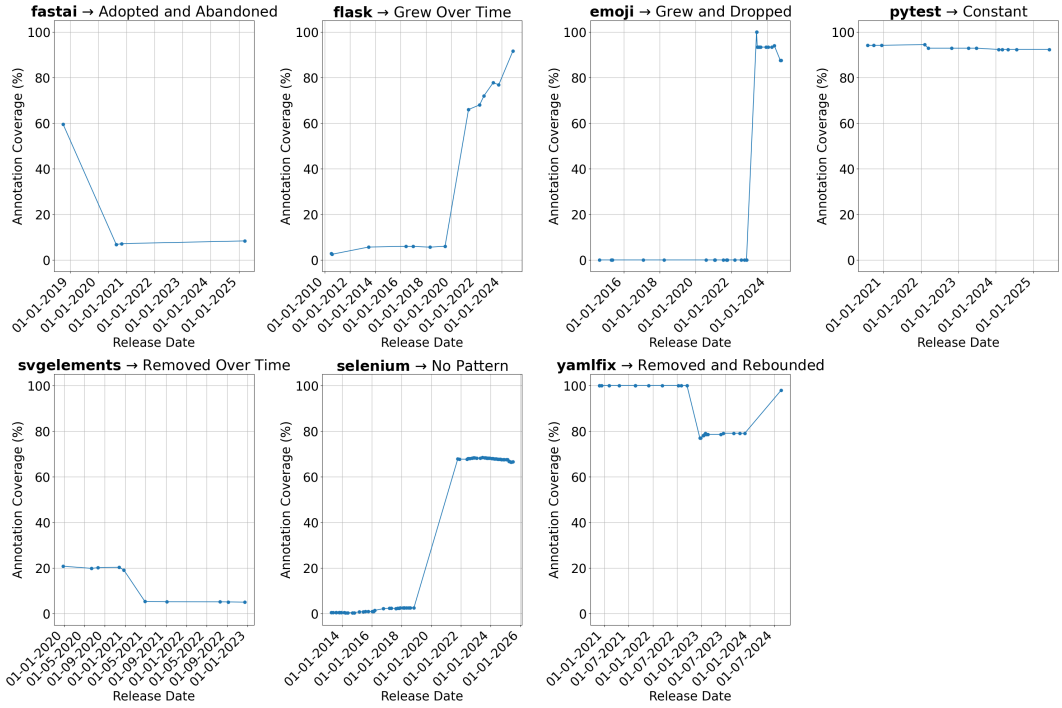

Figure 3: Evolution trends of representative libraries illustrating each adoption pattern.

Table 3: Distribution of libraries across annotation adoption evolution patterns.

Pattern	Number of Libraries
Adopted and Abandoned	207 (4.13%)
Grew over Time	885 (17.65%)
Grew and Dropped	45 (0.90%)
Removed over Time	194 (3.87%)
Removed and Rebounded	21 (0.42%)
Constant	3,004 (59.91%)
No Pattern	658 (13.12%)
Total Libraries	5,014 (100%)

removed annotations (e.g., svgelements [55]), and interestingly, 0.42% rebounded after initially starting to remove them (e.g., yamlfix [29]). Finally, our quantitative analysis could not detect obvious patterns for 13.12% of the libraries (e.g., selenium [44]). We further examine these cases in our qualitative analysis.

Qualitative Results. We find that the commits/Issues/PRs changing annotations did not provide rationale for the majority of the packages. We find explicit rationales for adding annotations in only 15 repositories. We discuss the common rationales with examples:

- **Linting:** In *icmplib* and *Snakemake*, pull requests cited type annotations for better linting and editor support; one *Snakemake* PR added type hints for some classes to fully utilize modern editor code prompts [20].
- **Documentation:** In *Squidpy* and *syspathmodif*, type hints replaced type information previously in docstrings. A recent issue in *syspathmodif* stated *"Docstrings indicate the type of parameters and return values. Instead, type hints must fulfill that role."* This issue was closed via a PR adding type hints [15].
- **Fixing static errors:** In *Scalene*, type hints were added to "mollify," "pacify," and "appease" *mypy*, reflecting reactive adoption [1]. In *Kanjize*, an open issue prompted a fix for a *Pylance* mismatch error, though the code ran correctly [60]
- **Code clarity:** In *wc-api-python* and *Snakemake*, developers introduced type hints "for better development experience" [36] or "to understand the logging system better" [24], highlighting their role in improving readability and maintainability.

Explicit reasons for removing type hints were rare. In *Scalene*, a type hint was removed for causing an error [52], while in *Blacksmith*, hints were dropped for being unnecessary [30].

On the other hand, we do see that library users sometimes requested annotations: in *MXNet*, to prevent bugs [23], and in *XBlock*, to improve code clarity in its abstract design [31]. Type hints also revealed overlooked design flaws. A *Squidpy* developer remarked, *"Thanks! I didn't look closely enough; more reasons why types are useful,"* [3] acknowledging how type annotations can clarify code. In other cases, missing hints or autocompletion highlighted API misuse, showing that type annotations can guide correct usage.

Discussion. Our quantitative results reveal six patterns in annotation evolution. Prior work [7] reports that developers add annotations regularly, in a sprint-like effort, or occasionally. We also

observe that most libraries maintain relatively stable annotation coverage or show gradual growth over time. We also uncover several patterns not previously documented: some libraries initially grow in annotation coverage but later abandon up to 80% of their annotations, while others remove annotations gradually over time. While we could not find explicit reasons for why some libraries abandoned annotations, our qualitative analysis shows that some libraries adopt type annotations mainly to improve linting, code clarity, documentation, and fix static errors. It was interesting to see that type annotations also reveal blind spots, uncover bugs, and expose API misuse, reinforcing that type hints can enhance the client developer's experience in using the library.

RQ2 Summary: The majority of libraries (59.91%) maintain a constant level of annotation coverage over time, while 17.65% gradually grow their annotation coverage. Few libraries abandon annotation. Reasons for annotation adoption include improved linting, code clarity, documentation, and fixing static errors. Type hints can also reveal design flaws and client API misuse.

7 RQ3: Which functions do developers prioritise annotating?

Methods. In this RQ, we focus on the 9,190 packages that have adopted annotations in their latest release (See Table 1 from RQ1). For each package, we calculate the proportion of public functions that are annotated versus those of private functions. To further investigate which libraries prioritize public versus private functions, we compute a **prioritization ratio** for each library:

$$\textbf{prioritization\_ratio}(\mathbf{p}) = \frac{\text{percent of public functions annotated}}{\text{percent of private functions annotated}}$$

We then categorize libraries based on this ratio:

- ratio > 1: library prioritizes the annotation of public functions (*Public Prioritized*)
- ratio < 1: library prioritizes the annotation of private functions (*Private Prioritized*)
- ratio ≈ 1: library annotates public and private functions roughly equally (*Equal*)

Finally, we distinguish between type annotations of type `Any` and more specific types. Type hints are meant to communicate the expected input and output of a function, so using `Any` conveys little information to readers or IDEs. Moreover, static type checkers allow any operation on such functions, reducing their ability to detect type mismatches. Prior studies have shown that function arguments and return types are the most commonly annotated [7]. Therefore, for each library, we calculate the proportion of function parameters and return types that use `Any` and examine how these functions are distributed between public and private functions.

Results. Figure 4 compares the distribution of type annotation coverage between public and private functions across all 9,190 adopting libraries. We find that private functions show a higher median coverage (36.8%) compared to public functions (14.1%). This difference is statistically significant, as confirmed by the Mann–Whitney U test ($p = 3.29e-181$, one-sided).

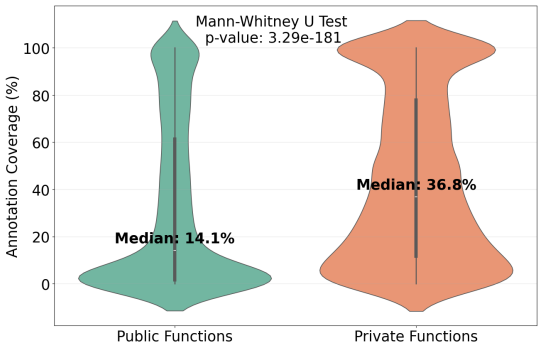

Figure 4: Coverage differences between private and public functions across all libraries.

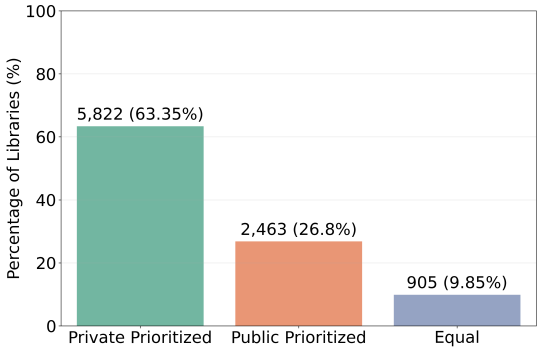

Figure 5: Distribution of function type prioritization

Examining the priority ratio further supports this observation. Figure 5 shows that 63.35% of libraries prioritize annotating private functions compared to only 26.8% that prioritize public functions. *pytest* is one example of a Public Prioritized package with $p_{\text{ratio}} = 93.30$.On the other side, *requests* is an example of a Private Prioritized package with $p_{\text{ratio}} = 0.11$. The remaining 9.85% of libraries annotate both categories equally, with *humanize* being one such example. Overall, our results indicate that libraries are more likely to annotate their private functions before their public ones.

We next look at the prevalence of the `Any` type in function annotations across both parameters and return types. Private parameters and return types rarely use `Any` annotations, with medians at 0%, though a few packages raise the means to 5.83% for parameters and 12.97% for returns. Public parameters and returns are slightly more annotated with `Any`, showing medians of 2.33% and 0% and means of 6.12% and 6.99%, respectively. Overall, `Any` annotations are rare, and the distributions are highly skewed, with a small number of packages driving most of the usage.

Discussion. Contrary to our initial expectations, library developers generally prioritize annotating private (internal) functions over public (client-facing) functions. This trend suggests that developers focus first on ensuring internal correctness, maintainability, and safe refactoring, before extending typing to functions exposed to external clients. Consequently, client-facing benefits such as improved IDE autocompletion, static type checking, and discoverability may only be realized later. The analysis of `Any` usage shows that it is rare for both parameters and return types. This indicates

that when library developers annotate, they use more specific types, enhancing type safety and the effectiveness of static type checkers.

RQ3 Summary: Developers tend to prioritize annotating private (internal) functions over public (client-facing) functions. Very few libraries achieve roughly equal annotation across public and private functions. The usage of `Any` is rare, indicating that when annotations are applied, developers use specific types.

8 Implications

We discuss the implications of our results for multiple stakeholders in the Python ecosystem.

Library developers. We find that the majority of libraries (83.39%) have adopted annotations, yet their average coverage remains below half (34.55%) of their functions. Moreover, 37.8% of these libraries have annotated no more than 10% of their functions. Our results provide an empirical motivation for library developers to continue improving annotation coverage rather than plateauing after adoption, as 59.91% of the libraries currently do. Additionally, to enhance IDE support and improve type safety for client developers, library maintainers should consider increasing annotation coverage for public functions.

Client developers. We observe that a small fraction (4.13%) of libraries have adopted and subsequently abandoned annotations. Client developers and users of these libraries may consider switching to functionally similar packages with a higher and more stable annotation adoption rate to benefit from improved static analysis and early bug detection. For users who cannot easily switch, we recommend checking for available stub files on Typeshed or PyPI that they can install to supplement the package with type information.

Community contributors. Companies such as Meta [34], which are committed to open source initiatives like PyTorch [33] and collaborate with Quansight [48] to advance type annotation support [9, 59], can use our dataset and findings to prioritize libraries with lower annotation coverage, which may strengthen the ecosystem's overall type completeness. At the ecosystem level, tools such as PyPI could display badges that indicate both annotation rates and the proportion of `Any` types based on our analysis, promoting transparency and encouraging broader adoption of optional typing. Finally, community contributors can improve the annotation coverage rate of low adopting libraries by submitting annotation stubs to Typeshed or by raising annotation-related pull requests in the respective library repositories.

Researchers. API misuse researchers could use our dataset to examine whether libraries with lower annotation coverage exhibit higher API misuse, as a prior study suggests the addition of type annotations as a potential mechanism to reduce API misuse [19].

9 Threats to Validity

Construct Validity. When measuring evolution patterns, we consider only major and minor releases. This decision may result in a smaller subset of versions for some libraries, potentially missing annotation changes introduced in patch releases. However, this is unlikely to affect our findings because, according to the Semantic Versioning 2.0.0 specification [43], patches are typically reserved for backwards-compatible bug fixes. Even for packages that do not strictly follow semantic versioning, we would capture any annotation changes introduced in patch releases in the subsequent minor or major release that we include in our analysis.

Precisely identifying client facing APIs in Python is a challenge, because the distinction between public and private functions is not formally defined with access modifiers. We rely on Pyright's internal logic, which follows Python's naming conventions. Consequently, libraries that do not follow these conventions may be misclassified. Given that the packages in our dataset are marked as *stable*, *production ready*, or *mature*, we believe that it is unlikely that they do not follow these conventions.

Finally, we analyze libraries that include annotations bundled within their distributed package, as this reflects what end users and static analysis tools have access to upon installation. Annotations may also be available through separate stub packages or from the *Typeshed* repository. Even though it is rare for packages to distribute their annotations externally (only 0.2% of 688,284 PyPi packages have names starting with `types-` or end with the `-stubs`[5]), we acknowledge that our reported annotation coverage can be interpreted as a lower bound.

Internal Validity. We use Pyright[35] for both annotation extraction and the identification of public and private functions. Consequently, the accuracy of our analysis depends on Pyright's precision. Pyright is developed by Microsoft and has been empirically shown to achieve the highest recall (74.55%) and F1 score (57.95%) among major Python type checkers such as Pyre, Pytype, and MyPy in detecting type-related errors [28]. This implies that Pyright is comparatively effective in identifying type annotations. Moreover, Pyright's `–verifytypes` feature has been adopted in large-scale community efforts to raise NumPy's type-completeness score from 33% to nearly 90% [14].

External Validity. We focus on Python and third-party libraries/-packages, and our results may not generalize to other dynamic languages such as JavaScript.

10 Conclusion

In this paper, we conduct an empirical study to understand how third-party libraries on Python Package Index (PyPI) use type annotations. We analyzed 76,327 versions across 11,021 libraries. Our analysis shows that most libraries (83.39%) have adopted annotations, with a median coverage of 19.68%. We identified six distinct annotation adoption patterns, with 59.91% of libraries maintaining stable annotation coverage and 17.65% showing gradual growth. We observed developers adopting type annotations to improve code clarity, linting, documentation, and static analysis but may remove annotations if they lead to more errors. However, fewer than 10% of libraries removed annotations after adopting them. We also found that developers tend to prioritize annotating private functions.

We recommend that client developers, library maintainers, and community contributors collaborate to increase type annotation coverage across the Python ecosystem. Researchers, on the other

[5]We query the names of all packages using the PyPI simple API: https://pypi.org/simple/

hand, can use our dataset as a foundation for future work, such as investigating whether clients of libraries with lower annotation coverage exhibit higher API misuse.

11 Acknowledgements

We acknowledge use of GenAI (GitHub Copilot, Cursor, ChatGPT, and Gemini) for assistance with coding parts of our data analysis scripts, as well as for grammar and spelling editorial support.

References

[1] Emery Berger. 2024. Types to mollify mypy; fixed an apparent typo-bug in signal handling. https://github.com/plasma-umass/scalene/pull/825. Merged May 17, 2024.
[2] Stephen Cass. 2025. The Top Programming Languages 2025. https://spectrum.ieee.org/top-programming-languages-2025 Accessed: 2025-10-14.
[3] chaichontat. 2022. Add type stubs to datasets. https://github.com/scverse/squidpy/pull/480. Merged March 16, 2022.
[4] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. 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) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 83–93. https://doi.org/10.1145/3387904.3389253
[5] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 87, 13 pages. https://doi.org/10.1145/3597503.3639184
[6] William G. Cochran. 1977. *Sampling Techniques* (3 ed.). John Wiley & Sons, New York.
[7] Luca Di Grazia and Michael Pradel. 2022. The evolution of type annotations in python: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3540250.3549114
[8] PyPI Docs. [n. d.]. PyPI BigQuery API Documentation. https://docs.pypi.org/api/bigquery/. Accessed: 2025-10-22.
[9] Meta Engineering. 2025. *Enhancing the Python ecosystem with type checking and free threading*. https://engineering.fb.com/2025/05/05/developer-tools/enhancing-the-python-ecosystem-with-type-checking-and-free-threading/ Accessed: 2025-10-14.
[10] Facebook. 2018. Pyre: Static Type Checker for Python. https://pyre-check.org/.
[11] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 758–769. https://doi.org/10.1109/ICSE.2017.75
[12] Google. 2012. Pytype: Static Type Analyzer for Python. https://github.com/google/pytype. Infers types without requiring annotations; support until Python 3.12.
[13] Google Cloud. [n. d.]. BigQuery. https://cloud.google.com/bigquery.
[14] Marco Gorelli and Quansight Labs. 2025. Bringing NumPy's type-completeness score to nearly 90%. Blog post, Pyrefly. https://pyrefly.org/blog/numpy-type-completeness/ Accessed: 2025-10-14.
[15] GRV96. 2025. Type hints. https://github.com/GRV96/syspathmodif/issues/22. Opened 2025-09-16.
[16] Yimeng Guo, Zhifei Chen, Lin Chen, Wenjie Xu, Yanhui Li, Yuming Zhou, and Baowen Xu. 2024. Generating Python Type Annotations from Type Inference: How Far Are We? *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 123 (June 2024), 38 pages. https://doi.org/10.1145/3652153
[17] Stefan Hanenberg. 2010. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. *SIGPLAN Not.* 45, 10 (Oct. 2010), 22–35. https://doi.org/10.1145/1932682.1869462
[18] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Softw. Engg.* 19, 5 (Oct. 2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1
[19] Xincheng He, Xiaojin Liu, Lei Xu, and Baowen Xu. 2023. How Dynamic Features Affect API Usages? An Empirical Study of API Misuses in Python Programs. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 522–533. https://doi.org/10.1109/SANER56733.2023.00055
[20] Hocnonsense. 2023. Add type hints to core codebase. https://github.com/snakemake/snakemake/pull/2375.
[21] Jeremy Howard and Sylvain Gugger. 2017. fastai: A Layered API for Deep Learning. https://github.com/fastai/fastai. GitHub repository.
[22] Md. Hussain and Ishtiak Mahmud. 2019. pyMannKendall: a python package for non parametric Mann Kendall family of trend tests. *Journal of Open Source Software* 4, 39 (25 7 2019), 1556. https://doi.org/10.21105/joss.01556
[23] jaanli. 2017. Python stubs for static type checking. https://github.com/apache/mxnet/issues/8359. Opened October 20, 2017.
[24] jlumpe. 2025. Add type annotations in logging module. https://github.com/snakemake/snakemake/pull/3757. Merged October 1, 2025.
[25] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. 2022. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3145–3158. https://doi.org/10.1109/TSE.2021.3082068
[26] Taehoon Kim. 2014. emoji: Emoji terminal output for Python. https://github.com/carpedm20/emoji. GitHub repository.
[27] Leslie Kish. 1965. *Survey Sampling*. John Wiley & Sons, New York.
[28] Xinrong Lin, Baojian Hua, Yang Wang, and Zhizhong Pan. 2023. Towards a Large-Scale Empirical Study of Python Static Type Annotations. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 414–425. https://doi.org/10.1109/SANER56733.2023.00046
[29] lyz code. 2020. yamlfix: A simple opinionated yaml formatter that keeps your comments. https://github.com/lyz-code/yamlfix. GitHub repository.
[30] Mardiros. 2022. Refactor typing annotations. https://github.com/mardiros/blacksmith/commit/3ab2aed8f24e134db23ef194bdebaf94630f63af. Commit 3ab2aed8f24e134db23ef194bdebaf94630f63af.
[31] Kyle McCormick. 2024. Add type hints and check them in CI. https://github.com/openedx/XBlock/issues/707. Opened January 12, 2024.
[32] Meta (Facebook). 2025. Pyrefly: Fast Static Type Checker and IDE for Python. https://github.com/facebook/pyrefly. Version 0.36.0, released October 2025.
[33] Meta Open Source. 2025. PyTorch. https://opensource.fb.com/projects/pytorch. Accessed: 2025-10-14.
[34] Meta Platforms, Inc. 2025. Meta Platforms, Inc. https://about.meta.com. Accessed: 2025-10-14.
[35] Microsoft. 2019. Pyright: Static Type Checker for Python. https://github.com/microsoft/pyright.
[36] MilmanRonV. 2024. Added Response type hint to API's request method. https://github.com/woocommerce/wc-api-python/pull/81. Merged May 17, 2024.
[37] Mypy Developers. 2012. Mypy: Optional Static Typing for Python. http://mypy-lang.org/.
[38] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software* 161 (2020), 110460. https://doi.org/10.1016/j.jss.2019.110460
[39] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 922–934. https://doi.org/10.1145/3540250.3549130
[40] Wonseok Oh and Hakjoo Oh. 2024. Towards Effective Static Type-Error Detection for Python. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1808–1820. https://doi.org/10.1145/3691620.3695545
[41] Pallets. 2010. Flask: The Python micro framework for building web applications. https://github.com/pallets/flask. GitHub repository.
[42] L. D. Paulson. 2007. Developers shift to dynamic programming languages. *Computer* 40, 2 (Feb 2007), 12–15. https://doi.org/10.1109/MC.2007.53
[43] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. https://semver.org/. Accessed: 2025-10-15.
[44] Selenium Project. 2008. Selenium: A browser automation framework and ecosystem. https://github.com/SeleniumHQ/selenium. GitHub repository.
[45] pytest dev. 2010. pytest: the pytest testing framework. https://github.com/pytest-dev/pytest. GitHub repository.
[46] Python Software Foundation. [n. d.]. Python Package Index (PyPI). https://pypi.org/.
[47] Python Typeshed Contributors. 2025. Typeshed: Collection of Python Library Stubs. https://github.com/python/typeshed.
[48] Quansight. 2025. Quansight. https://quansight.com/. Accessed: 2025-10-14.
[49] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. 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) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3426422.3426981
[50] Ethan Smith, Jukka Lehtosalo, and Michael Sullivan. 2018. PEP 561 – Distributing and Packaging Type Information. https://peps.python.org/pep-0561/. Python Enhancement Proposal 561.
[51] Stack Overflow. 2025. Technology | 2025 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025/technology/#most-popular-technologies. Accessed: 2025-10-14.
[52] Sam Stern. 2021. Removed type hint because was causing an error. https://github.com/plasma-umass/scalene/

commit/ea560ff5d48cbf22976ddb4110df0d9ac850d4c8. Commit ea560ff5d48cbf22976ddb4110df0d9ac850d4c8.

[53] Li Sui, Shawn Rasheed, Amjed Tahir, and Jens Dietrich. 2022. A study of single statement bugs involving dynamic language features. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 494–498. https://doi.org/10.1145/3524610.3527883

[54] Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. 2023. Static Type Recommendation for Python. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 98, 13 pages. https://doi.org/10.1145/3551349.3561150

[55] Tatarize. 2019. svgelements: SVG Parsing for Elements, Paths, and other SVG Objects. https://github.com/meerk40t/svgelements. GitHub repository.

[56] TIOBE Software. 2025. TIOBE Index for October 2025. https://www.tiobe.com/tiobe-index/. Accessed: 2025-10-14.

[57] Guido Van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 - Type Hints. https://peps.python.org/pep-0484/. Python Enhancement Proposal.

[58] Wenjie Xu, Lin Chen, Chenghao Su, Yimeng Guo, Yanhui Li, Yuming Zhou, and Baowen Xu. 2023. How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 242–253. https://doi.org/10.1109/ICPC58990.2023.00039

[59] Danny Y. 2025. Call For Suggestions: Nominate Python Packages for Typing Improvements. https://discuss.python.org/t/call-for-suggestions-nominate-python-packages-for-typing-improvements/80186 Accessed: 2025-10-14.

[60] yahiro code. 2025. Return type may be incorrect. https://github.com/nagataaaas/Kanjize/issues/9. Opened May 3, 2025.

[61] Asimina Zaimi, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos. 2015. An Empirical Study on the Reuse of Third-Party Libraries in Open-Source Software Development. In *Proceedings of the 7th Balkan Conference on Informatics Conference* (Craiova, Romania) *(BCI '15)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2801081.2801087

[62] Jelle Zijlstra. 2024. PEP 749 – Implementing PEP 649. https://peps.python.org/pep-0749/. Python Enhancement Proposal 749.