Characterizing Python Method Evolution with *PyMevol*: An Essential Step Towards Enabling Reliable Software Systems

Haowei Quan, Jiawei Wang, Bo Li, Xiaoning Du
Monash University
Melbourne, Australia
{Haowei.Quan, Jiawei.Wang1, Xiaoning.Du}@monash.edu,
limber0117@gmail.com

Kui Liu
Huawei
Hangzhou, China
brucekuiliu@gmail.com

Li Li
Monash University
Melbourne, Australia
Li.Li@monash.edu

***Abstract*—Understanding the evolution of library methods is essential for maintaining high-quality and reliable software systems as those libraries often evolve rapidly in order to meet new requirements such as adding new features, improving performance, or fixing vulnerabilities. Failing to incorporate this evolution may result in compatibility issues that may manifest themselves as runtime crashes, leading to a poor user experience. This is not uncommon for the most popular programming language, Python, for which our community has developed over 380,000 libraries. To help developers better understand their used libraries, we propose to the community a prototype tool called *PyMevol* to model Python libraries' APIs and their evolution. Specifically, given a Python library, *PyMevol* statically examines its code to extract APIs (including aliases introduced by Python's import-flow mechanism) from all its released versions to build a history-sensitive alias-aware API explorer tree, a tree structure that allows users to explore the biography of each API so as to quickly locate where and when a given API is introduced, changed, or removed. Our experimental results over five popular real-world Python libraries show that our approach is reliable in achieving its purpose (i.e., over 90% of accuracy) and helpful in supporting further API-relevant analyses.**

***Index Terms*—Python, API evolution, static analysis**

I. Introduction

Python has gained increasing popularity in recent years. According to IEEE Spectrum[1], Python has become the most popular language since 2021 by overtaking Java and C, which have dominated software production for decades. One reason that makes Python the most popular programming language could be the large number of Python libraries made readily available by the Python community. Indeed, there are over 380,000 libraries in the Python Package Index (PyPI) repository. Each library further supplies hundreds (or even thousands) of reusable functions (known as Application Programming Interfaces, or APIs in short) that hide implementation details for facilitating Python application development.

Unfortunately, the software analysis community has not yet caught up with the popularity of the Python language per se. Currently, there have not been many works [1], [2] proposed to help the community develop reliable Python applications. As argued by Yang et al. [3], even in 2022, static analysis tooling for Python is not yet widely developed or used, while such tooling will undoubtedly benefit Python developers to achieve reliable software systems.

To fill this gap, as our initial attempt, we propose an automated approach to characterize the API evolution of Python libraries. We believe this is essential for enabling reliable software development. Indeed, Python applications often rely on third-party libraries to achieve their objectives, and the libraries will be inevitably evolved for fixing defects, bugs, vulnerabilities, and adding new features, etc. Such an evolution may lead to breaking changes (e.g., removed APIs) that subsequently will impact the reliability of its client applications. A comprehensive understanding of method evolution would help mitigate such impacts.

Actually, our community has acknowledged the merits of software evolution studies and thereby proposed various approaches to characterize the evolution of software systems. For example, Li et al. [4] have presented a tool, CDA, for characterizing the evolution of Android APIs. Among various findings reported by the authors, representative ones include bug issues reported to the Android Open Source Project (AOSP) team and API usage problems (e.g., accessing inaccessible or incompatible ones [5], [6]) that have to be specifically handled by app developers for their apps accessing those APIs. Their experimental results echo our claim that understanding the software evolution indeed helps enable the reliability of software systems, being helpful for not only the studied object itself but also all its client applications.

Despite the fact that many efforts have been put into characterizing the evolution of software systems developed with other program languages, whether these observations and conclusions apply to Python applications are unknown until a firsthand exploration is conducted. The majority of works are primarily designed for studying statically typed languages, such as Java [5]–[14]. However, as argued by Yang et al. [3], because Python is a dynamical language, the existing approaches proposed for handling statically typed

* Li Li is the corresponding author.

[1]https://spectrum.ieee.org/top-programming-languages/

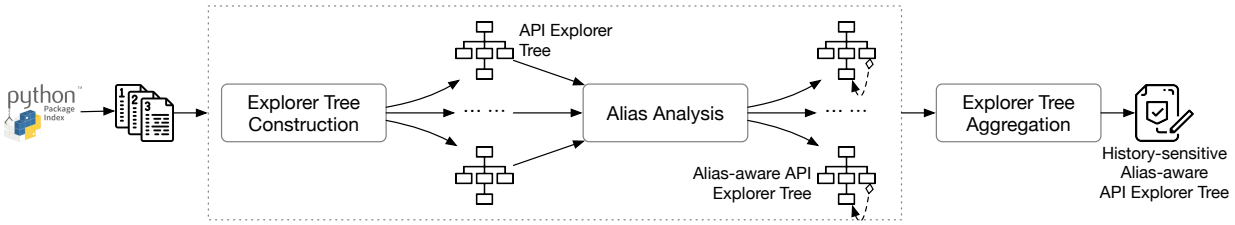


Fig. 1: The working process of *PyMevol*.

languages cannot be effectively applied to analyze Python code. Furthermore, as argued by Zhang et al. [1], extracting Python library APIs is non-trivial because Python embraces many advanced features to ease the development of Python applications. For example, Python's import mechanism allows library developers to create API aliases (mainly to create shorter names for neat references), and these aliases will be maintained consistently with the evolution of the directly defined ones. Referring to outdated API aliases can also cause compatibility issues in client applications, and such feature need to be carefully addressed to precisely characterize the effect of API evolution. In what follows, APIs refers to both directly defined APIs and their aliases.

In this paper, we propose a prototype tool, namely *PyMevol*, to support the characterization of method evolution in Python libraries. Specifically, *PyMevol* statically examines the historical code of a given library and constructs a historical-sensitive alias-aware API explorer tree, which records the lifecycle of each method (i.e., when introduced, updated, and removed) and its aliases (detailed in Section II-B). To demonstrate the effectiveness of *PyMevol*, we apply it to five popular Python libraries and the experimental results show that *PyMevol* can accurately capture the changes of library APIs. We further demonstrate *PyMevol*'s usefulness by leveraging *PyMevol* to conduct API usage analysis.

To summarize, our work makes the following two main contributions.

- We propose, *PyMevol*, to characterize method evolution in Python by statically building a history-sensitive alias-aware API explorer tree.
- We evaluate the effectiveness and usefulness of *PyMevol* based on five popular Python libraries and over 4,000 real-world Python projects.

II. Approach

Fig. 1 illustrates the working process of *PyMevol*, which takes as input a library and outputs a history-sensitive alias-aware API explorer tree that dedicatedly records the detailed evolution information of the input library. The historical information of the given library is extracted from the PyPI repository, and the alias information is identified through a detailed static analysis based on the import-flow relationships defined in the library code. The output tree is designed to include comprehensive information about the library. It is expected to be the default place for users to go for, when they are interested in understanding the evolution of certain APIs. The working process of *PyMevol* is mainly made up of three modules, i.e., (1) Explorer Tree Construction, (2) Alias Analysis, and (3) Explorer Tree Aggregation. We now detail these three modules, respectively.

A. Explorer Tree Construction

As the first module, *PyMevol* aims at constructing an API explorer tree for a given library. An API explorer tree is a tree data structure used to represent the composition structure of the library and store information of APIs, where each node is an explorer node, which is used to represent a package, a sub-package, a module, a class or an API.

There are three different explorer node types, each of which has different properties and stores different information.

- **Package/Module Node**: In Python, each source file is deemed as a module[2] and the file name (without extension) is regarded as the module name. A Python package is like a directory[3] holding sub-packages and modules. For the sake of simplification, in this work, we present all packages, sub-packages, and modules as Package/Module nodes.
- **Class Node**: A directly defined class is represented by a Class node, which stores the class name, the fully qualified name of the class, the source code of the class, and the reference to its alias nodes.
- **API Node**: A directly defined public method is recorded as an API Node, which stores the API name, the fully qualified name of the API, the parameter keywords, the default values of the parameters, the source code, and the reference to the aliases of the API. As Python does not support method overloading, we construct one API Node for one fully qualified API name. The API Nodes are the leaf nodes of the explorer tree and do not have child nodes. The parent of an API Node can be a Class node or a Package/Module node.

As shown in Fig. 1, given a library, *PyMevol* first extracts all its historical versions from the PyPI repository. Then, for each of the located versions, it statically analyzes the code to construct an API explorer tree, respectively. Particularly, *PyMevol* goes through all the Python files in the given library to record the overall file structure. For each of the visited Python files, *PyMevol* will build an Abstract Syntax Tree (AST) for it and will traverse the tree to identify all the defined properties (e.g., classes, methods, etc.).

Fig. 2 presents a simple example of an API explorer tree. Each path from the root node to a leaf node forms a fully qualified API name, which is unique for the library. To this end, we consider the fully qualified name as the signature of

[2]https://docs.python.org/3/reference/import.html

[3]It must have a file named *__init__.py* in order to be qualified for a Python package.

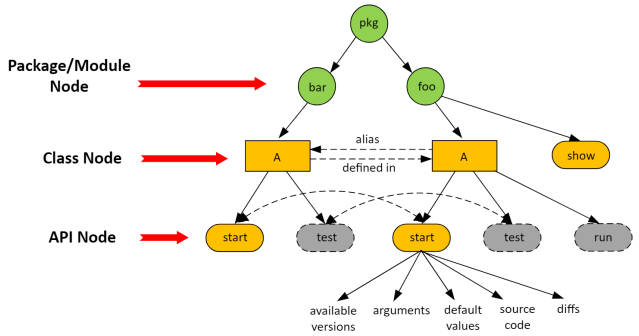


Fig. 2: An example of API explorer tree. The grey color indicates that the API is removed in the latest version of the library.

the given API. As shown in Fig. 2, for each node, we further connect its associated data (e.g., available versions, method's arguments) to increase the usefulness of the API explorer tree.

B. Alias Analysis

Python has an import mechanism called transitive import [15]. Specifically, if a method/class/module is imported in a module, such method/class/module will be added to the namespace of current module at runtime. For this reason, if Module A is imported in Module B which is imported in Module C, then Module A is accessible to Module C. It is very popular for Python libraries to use the transitive import mechanism to shorten the fully qualified names of required APIs by importing them in higher level modules. Take the library *Pandas*, one of the most popular tools for data scientists, as an example, the API *pandas.core.arrays.categorical.Categorical* is imported in *pandas.core.arrays* module, and hence forms an alias as *pandas.core.arrays.Categorical*.

In Python, a method/class/module can be transitively imported multiple times. Let us take the same API for example, *pandas.core.arrays.categorical.Categorical* has an alias as *pandas.Categorical*, which is an API for the Categorical class in *pandas*'s official documentation[4] and is generated by importing an alias *pandas.core.api.Categorical* in the top level module *pandas*. The mapping relationships between a directly defined API and all its aliases form an API alias map.

Python libraries make heavy use of the transitive import mechanism and have a significant number of aliases of APIs, which makes API analysis in Python much more complex than the others. Thus, we introduce into *PyMevol* a dedicated alias analysis module to properly model the import flows (based on the previously generated ASTs) so as to build the map connecting APIs to their aliases. Specifically, we first traverse the parsed ASTs of the source files and extract import information. We iteratively construct and update an import graph until it converges (the import graph does not change with more iterations) or reaches a maximum number of iterations. The maximum number of iterations can be empirically set to, e.g., 3, to seek a balance between time efficiency and completeness of *PyMevol*. Based on the import information in the import graph, we deduce the alias relationships.

[4]https://pandas.pydata.org/docs/reference/api/pandas.Categorical.html

TABLE I: A summary of Python libraries used for evaluation

Name	LOC	commits	#stars	Used by	Category	# APIs
TensorFlow	7m	132k	166k	202k	deep learning	49k
scikit-learn	455k	29k	51k	361k	data analytics	3k
Pandas	783k	30k	35k	731k	data analytics	43k
Django	984k	31k	65k	948k	web development	25k
Flask	35k	5k	60k	1m	web development	1k

To properly record aliases, we improve the aforementioned API explorer tree to be alias-aware by adding new Class Nodes and API Nodes for classes' aliases and APIs' aliases, respectively. We further add special edges to connect the aliases to their original definitions (cf. dashed edges in Fig. 2).

C. Explorer Tree Aggregation

Recall that *PyMevol* will generate one API explorer tree for every library version available in the PyPI repository. The last module hence aims at aggregating those independent trees into a single model, named the history-sensitive alias-aware API explorer tree. When merging a new version (represented by a single API explorer tree) into the aggregated explorer tree, non-existing nodes (e.g., new APIs) will be added to the aggregated tree, and existing nodes will be merged. To record the history of a given API, we add a new attribute for each node, namely *available versions*, to maintain the number of versions the class/API is available. For such APIs that are eventually removed from the library, we will still keep them in the explorer tree (represented as grey nodes) for easy references (the *available versions* attribute records when the API is no longer accessible in the history of the library). When merging an API Node of a new version into the existing API Node in the tree, we will also compute the differences between the source code of the API of the new version and the source code of the descendant version in the unified format [16] (cf. through the *diffs* attribute as illustrated in Fig. 2). Such information will be useful when studying the compatibility of given APIs.

III. Evaluation

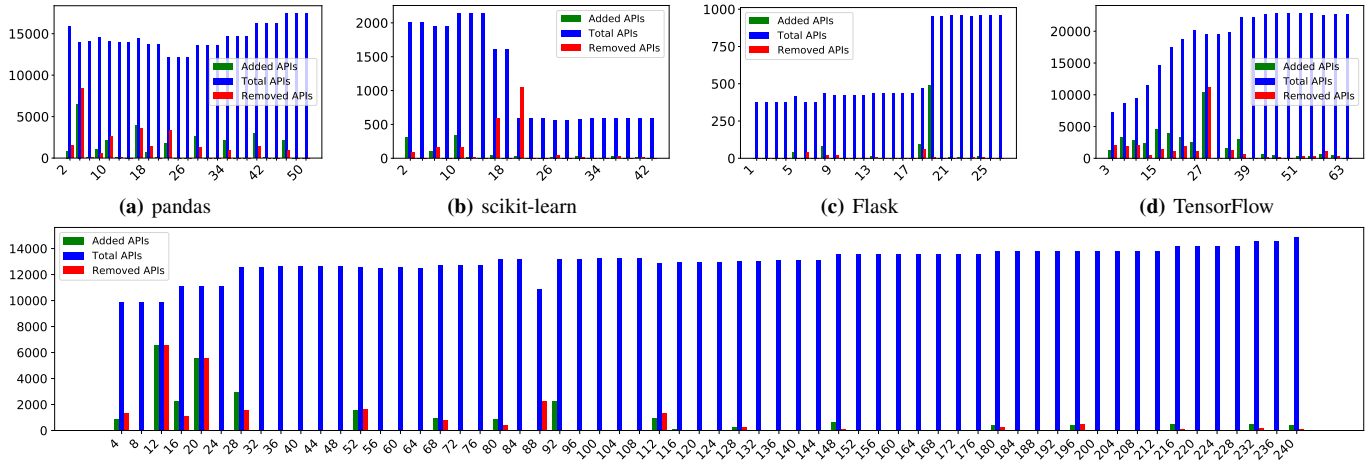
To evaluate the effectiveness and usefulness of *PyMevol*, we aim to experimentally answer the following three research questions.

- **RQ1:** How effective is *PyMevol* in supporting the characterization of API evolution in Python libraries?
- **RQ2:** How do API aliases evolve in Python libraries?
- **RQ3:** How useful is *PyMevol* in facilitating Python library analyses?

Dataset. To fulfill our experiments, we chose five Python libraries as our research subject. These five libraries are among the most popular ones and represent different software domains. Specifically, *TensorFlow* is a famous deep learning library, *scikit-learn* and *Pandas* are well-known data analytic libraries, *Django* and *Flask* are popular web development libraries. Detailed features are summarized in Table I.

A. API Evolution Characterization via PyMevol

To answer RQ1, we first investigate the correctness of *PyMevol* and then characterize the API evolution of the five Python libraries.



(a) pandas **(b)** scikit-learn **(c)** Flask **(d)** TensorFlow

(e) Django

Fig. 3: API evolution history. The X axis represents library versions (chronological order) while the Y axis is the amount of APIs.

TABLE II: Correctness of *PyMevol*

	TensorFlow	Django	Flask	scikit-learn	Pandas	Avg.
Accuracy	90%	92%	93%	93%	84%	90.4%

1) Accuracy of PyMevol*:* Considering the large number of APIs detected in those libraries, as shown in Table I, it is impossible to check if every API is correct manually. Therefore, given an API profile generated by *PyMevol*, we randomly sample 100 APIs and check if those APIs exist in the corresponding Python library. Specifically, we adopt the following two methods to check the existence of an API.

- **Automated Evaluation.** For module level APIs (e.g. pkg.foo.show), we attempt to import the API via *exec('from pkg.foo import show')*. If any error occurs, we consider the corresponding API does not exist. For class level APIs (e.g., pkg.foo.A.start), we first attempt to import the class by *exec('from pkg.foo import A')*. Then, we employ *hasattr*, *getattr*, and *callable* functions via *exec('check = hasattr(A, start) and callable(getattr(A, start, None))')* to checks if *start* indeed exists in *A*.
- **Manual Confirmation.** Unfortunately, we found that the above automatic validation may fail due to configuration errors or irrelevant import errors (i.e., dependency not specified by the library). Here, we further validate the failed APIs manually to check if they indeed exist in the library.

Table II summarizes the experimental results. We can find that *PyMevol* is capable of extracting APIs from Python libraries with an average accuracy of 90.4%. This high accuracy illustrates the practical usability of our approach. We further investigated those failed APIs and found that they belong to class methods transformed into 1) Python's property objects by "@property" decorator and 2) customized property objects using customized decorators. *Property* is a unique feature of Python used to take the responsibility of *getter*, *setter*, and *deleter* methods of a class attribute. *PyMevol* does not check the decorators of functions and hence has overlooked those APIs. We plan to improve *PyMevol* to handle decorators in our future work.

TABLE III: API statistics of the Python libraries

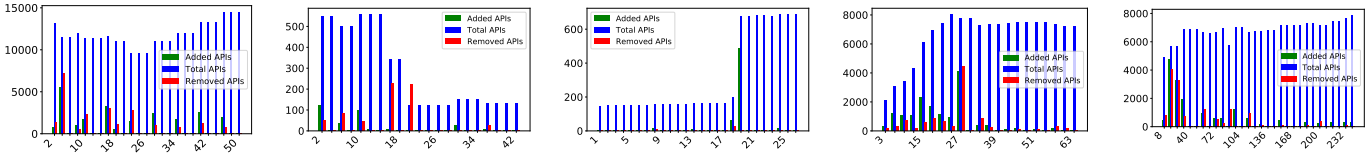
Library	# Versions	# APIs	# Added APIs	% Versions with New APIs	# Removed APIs	% Versions with Removed APIs
TensorFlow	65	49,055	40,981	30/65(46.2%)	26,304	27/65 (41.5%)
Django	241	24,579	14,252	145/241(60.2%)	9,699	70/241 (29.1%)
Flask	27	1,081	708	10/27(37.0%)	118	9/27 (33.3%)
scikit-learn	43	2,742	737	22/44(50.0%)	2,156	14/44 (31.8%)
Pandas	51	42,637	26,739	36/51(70.6%)	25,137	24/51 (47.1%)

2) API Evolution Characterization: We further investigate the evolution history of the libraries with the help of the API profiles. As introduced earlier, an API profile records the lifecycle of each API in the library, it is easy for us to characterize the API usage and evolution. Table III presents the API statistics of the five Python libraries based on their API profiles generated by *PyMevol*. Specifically, we extract the statistics with regard to the following questions: How many APIs have ever existed in the libraries? How many APIs have been removed/added? How frequent are APIs removed/added in the libraries? The removed/added APIs are identified by comparing the API lists of consecutive releases.

It can be observed that the API evolution patterns vary greatly among different libraries. For example, *TensorFlow* has only 65 different versions but has 26,304 removed APIs and 40,981 newly added APIs in total. In contrast, *Flask* released 27 versions with only 118 APIs removed and 708 APIs added. This indicates that developers using *TensorFlow* in their client applications may need extra efforts to deal with compatibility and reliability issues brought by the update of *TensorFlow*.

We can also find that API removals frequently happen in every Python library. For example, 41.5% of *TensorFlow* releases have API removals while 33.3% of *Flask* releases have API removals. This observation aligns with the findings reported recently by Wang et al [15]. Furthermore, the additions of the APIs happen more frequently than removals.

Fig. 3 presents the API evolution history of each Python library. We can find that different library may have very different evolution patterns. For example, while the total number of APIs in *Django* keeps increasing, the amount of APIs in *scikit-learn* drops significantly after the 20th update. We can also observe *TensorFlow* increases its APIs rapidly in early stage, and after version 36, the number of APIs starts to become stable.



(a) Pandas **(b)** scikit-learn **(c)** Flask **(d)** Tensorflow **(e)** Django

Fig. 4: Alias API evolution history. X-axis represents different library versions while Y-axis is the number of alias APIs.

TABLE IV: Alias API statistics of the Python libraries

Library	# Alias APIs	# Additions	% Versions with New APIs	# Removals	% Versions with API Removals
TensorFlow	16,624 (33.9%)	14,355	25/241(10.4%)	9,385	26/65 (40.0%)
Django	14,051 (57.2%)	8,796	128/241(53.1%)	6,153	60/241 (24.9%)
Flask	738 (68.3%)	591	9/27(33.3%)	49	7/27 (25.9%)
scikit-learn	794 (29.0%)	247	15/44(34.1%)	662	12/44 (27.3%)
Pandas	36,150 (84.8%)	22,966	34/51(66.7%)	21,617	22/51 (43.1%)

B. RQ2: Alias API Evolution Characterization

Due to the unique transitive import mechanism in Python, excessive alias APIs exist in the libraries. Now we investigate *PyMevol*'s ability to characterize alias API evolution.

Table IV presents the statistics of alias APIs detected by *PyMevol* from the five Python libraries. Firstly, we can find that the alias APIs exist widely in Python libraries. For example, 33.9% of APIs in *TensorFlow* are alias APIs and as many as 84.8% APIs in *Pandas* are alias APIs. Secondly, by comparing Table III to Table IV, we can find that the update of alias APIs has similar tendency to the update of overall APIs, which indicates the main cause of the additions/removals of aliases may be the additions/removals of directly defined APIs. This observation is further evidenced by Fig. 4 in which the alias API evolution history of each Python library is displayed according to different released versions.

Note that given a directly defined API, all its alias APIs are recorded in its alias map. Through a deep inspection of those alias maps, we find that the alias API evolution can be divided into 4 categories as follows.

- **Addition Only (AO)**. In a new version, only new aliases are added in the alias map or a new alias map is added.
- **Removal Only (RO)**. In a new version, some aliases are removed in the alias map or the entire alias map is removed.
- **Addition and Removal (AR)**. In a new version, the alias map not only adds new alias APIs but also removes existing alias APIs.
- **Directly Defined API Change (DC)**. A directly defined API is refactored while part or all of its alias APIs remain.

As shown in Table V, AO and RO dominate the alias API evolution for most of the Python libraries. This conforms to the finding that main cause of additions/removals may be due to the additions/removals of directly defined APIs.

TABLE V: Characterizing alias evolution pattern

Library	# AO	# RO	# AR	# DC
TensorFlow	13,113	10,987	440	487
Django	7,901	6,559	579	281
Flask	192	46	3	1
scikit-learn	243	484	14	15
Pandas	3,692	3,843	3,681	644

C. RQ3: Ability to Facilitate Library Analysis

In the last research question, we made an initial attempt to study *PyMevol*'s potential to facilitate Python library analysis in practice. More usage of *PyMevol* can be explored by the community in the future. While Python has excessive number of libraries available for developers, it would be beneficial to learn the API usage patterns in the community. The potential usefulness of such study include, but are not limited to, API discovery, API recommendation, API composition, API optimization, etc. Here, we present an exemplar study on API usage to demonstrate how it contributes to our community.

TABLE VI: Statistical results of client usage analysis

Library	# Used APIs	# Used Removed APIs	# Affected Projects
TensorFlow	2,408/49,055 (4.9%)	879/26,304 (3.3%)	248/922 (26.9%)
Django	1,797/24,579 (7.3%)	408/9,699 (4.2%)	405/957 (25.9%)
Flask	64/1,081 (5.9%)	15/118 (12.7%)	87/952 (9.1%)
scikit-learn	624/2,742 (22.8%)	385/2,156 (17.9%)	217/808 (26.9%)
Pandas	1,358/42,637 (3.2%)	547/25,137 (2.2%)	177/828 (21.4%)
Total	6,251/120,094 (5.2%)	2234/63,414 (3.5%)	977/4,467 (20.9%)

First, for each of the five Python libraries, we collect a total number of 1,000 best-match-ranked client applications returned by the GitHub search: the library name is used as the query term and Python is used as the development language. Next, we conduct a further scan to filter out applications that indeed do not use any of such libraries. This leaves us 4,467 client projects in total for evaluation, including 922 applications using *TensorFlow*, 957 applications using *Django*, 952 applications using *Flask*, 808 applications using *scikit-learn*, and 828 applications using *Pandas*. Finally, we extracted the API usage information of the five Python libraries from those client applications.

Table VI shows the statistical results of the analysis. The first observation is that although each Python library has a large number of APIs, only a limited portion of them are actually used by the community. Only 5.2% of APIs, on average, are used by the top-ranked client applications. This phenomenon is potentially beneficial to library developers. For example, given the API popularities in their libraries, developers can devote more effort to maintaining those popular APIs. In addition, it can help developers determine the priority of new APIs, e.g., mutants of popular APIs can be assigned a high priority in the development plans.

We further find that 20.9% of 4,467 client applications still use the removed APIs in their latest commit, and 2,234 of 63,414 removed APIs are used. This will inevitably lead to bugs and undermine the reliability of those applications. Please note that we only check the usage of removed APIs in the experiments. If we further include the changes in API parameters, the percentage of affected applications would be even higher. The observation also confirms that *PyMevol* is useful for enabling reliable software systems.

IV. Related work

Python Library Studies. Our work is closely related to Python library studies. Recent researches in this field mainly focus on library API usage [1], [2] and dependency analysis [15], [17]–[19]. To name a few, Zhang et al. [1] investigated the evolution patterns of Python libraries and detected compatibility issues. Wang et al. [2] proposed *dlocator* to locate the usage of deprecated APIs in client applications. The authors further proposed *SnifferDog* [15] to automatically restore the execution environment of Jupyter notebooks based on API usages analysis and pre-build API bank. Different from existing studies, we proposed *PyMevol* to characterize method evolution in Python by statically building a history-sensitive alias-aware API explorer tree.

Python Code Analysis. The advances in code analysis for Python are mostly on type inference [20], [21], static call graph construction [22], and analyzing code quality for Python. For instance, Wang et al. [23], [24] explore to detect unused variables, deprecated APIs and dynamically test the reproducibility of Python code snippets in Jupyter notebooks. Furthermore, He et al. present the first work on real time API recommendation for Python developers and Yi et al. [3] report the patterns of how complex Python language features such as functional features used by developers.

Framework Evolution Analysis. Our work is also related to the traditional framework evolution analysis. In this field, many studies have been conducted to identify the framework evolution in statically typed languages, such as Java [6]–[14]. However, we do not observe any efforts achieving the same purpose for Python frameworks.

Empirical studies have also been conducted to study the framework evolution [4], [5], [25]. Li et al. [4] performed an empirical study on API deprecation in Android framework evolution. Xavier et al. [25] presented a large-scale study to analyze the impact of breaking API changes in Java frameworks.

The essential step for framework evolution analysis is API extraction, which is difficult for Python due to its dynamically typed nature and advanced features [22].

V. Conclusion

This paper presents *PyMevol* – a static analysis tool to support the characterization of method evolution in Python. We evaluate the correctness of *PyMevol* and demonstrate that it is effective in supporting evolutionary studies of Python library APIs. We further illustrate the usefulness of *PyMevol* by conducting an empirical investigation of API usage analysis, which provides interesting insights that can be explored further in supporting reliable software analyses.

References

[1] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do Python framework APIs evolve? An exploratory study," in *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 81–92.

[2] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 233–244.

[3] Y. Yang, M. Fazzini, and M. Hirzel, "Complex Python features in the wild?" in *the 19th International Conference on Mining Software Repositories (MSR 2022)*, 2022.

[4] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "CDA: Characterising deprecated Android APIs," *Empirical Software Engineering (EMSE)*, 2020.

[5] L. Li, T. F. Bissyandé, Y. Le Traon, and J. Klein, "Accessing inaccessible Android APIs: An empirical study," in *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[6] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "CiD: Automating the detection of API-related compatibility issues in Android apps," in *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[7] L. Li, T. F. Bissyandé, and J. Klein, "Moonlightbox: Mining android api histories for uncovering release-time inconsistencies," in *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, 2018.

[8] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *the 30th International Conference on Software Engineering (ICSE)*, 2008, pp. 471–480.

[9] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[10] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: A hybrid approach to identify framework evolution," in *the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010, pp. 325–334.

[11] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 353–363.

[12] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

[13] M. Lamothe and W. Shang, "Exploring the use of automated API migrating techniques in practice: An experience report on Android," in *the 15th International Conference on Mining Software Repositories*, 2018, pp. 503–514.

[14] K. Huang, B. Chen, L. Pan, S. Wu, and X. Peng, "REPFINDER: Finding replacements for missing APIs in library update," in *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 266–278.

[15] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter notebooks," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1622–1633.

[16] GNU Diffutils. Detailed description of unified format. [Online]. Available: https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html

[17] E. Horton and C. Parnin, "Dockerizeme: Automatic inference of environment dependencies for python code snippets," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 328–338.

[18] ——, "V2: Fast detection of configuration drift in Python," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 477–488.

[19] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-based environment dependency inference for Python programs," in *ICSE 2022*, 2022, pp. 1245–1256.

[20] A. M. Mir, E. Latoškinas, S. Proksch, and G. Gousios, "Type4Py: Practical deep similarity learning-based type inference for Python," in *ICSE 2022*. ACM, 2022, p. 2241–2252.

[21] Y. Peng, Z. Li, C. Gao, B. Gao, D. Lo, and M. Lyu, "HiTyper: A hybrid static type inference framework with neural prediction," *arXiv preprint arXiv:2105.03595*, 2021.

[22] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical call graph generation in Python," in *ICSE 2021*. IEEE, 2021, pp. 1646–1657.

[23] J. Wang, L. Li, and A. Zeller, "Better code, better sharing: on the need of analyzing jupyter notebooks," in *ICSE-NIER 2020*, 2020, pp. 53–56.

[24] J. Wang, T.-Y. KUO, L. Li, and A. Zeller, "Assessing and restoring reproducibility of jupyter notebooks," in *ASE 2020*, 2020, pp. 138–149.

[25] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *SANER 2017*. IEEE, 2017, pp. 138–147.