Exploring How Deprecated Python Library APIs are (Not) Handled

Jiawei Wang
Li Li
{jiawei.wang1,li.li}@monash.edu
Monash University
Australia

Kui Liu
kui.liu@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics
China

Haipeng Cai
haipeng.cai@wsu.edu
Washington State University
USA

ABSTRACT
In this paper, we present the first exploratory study of deprecated Python library APIs to understand the status quo of API deprecation in the realm of Python libraries. Specifically, we aim to comprehend how deprecated library APIs are declared and documented in practice by their maintainers, and how library users react to them. By thoroughly looking into six reputed Python libraries and 1,200 GitHub projects, we experimentally observe that API deprecation is poorly handled by library contributors, which subsequently introduce difficulties for Python developers to resolve the usage of deprecated library APIs. This empirical evidence suggests that our community should take immediate actions to appropriately handle the deprecation of Python library APIs.

CCS CONCEPTS
• Software and its engineering → Software evolution.

KEYWORDS
Deprecated API, Deprecation, Python library, Evolution.

ACM Reference Format:

1 INTRODUCTION
Application Programming Interface (API) offers a communication protocol for different programs to interact with each other. Nowadays, APIs have often been regarded as the standard means providing interfaces of modern Software Development Kits (SDKs) or libraries to support and simplify the development of software. When using APIs, developers do not necessarily need to understand the underlying implementations of APIs that could be super complicated and could involve interacting with many other APIs. Just like the fast-evolving nature of software, e.g., for providing new features or fixing bugs and vulnerabilities, APIs need to be continuously maintained and updated as well [30]. Indeed, the implementation of APIs can also come with bugs or vulnerabilities that, if not being fixed, can propagate the same problems to their consumers, i.e., the software that accessed these APIs [61]. Since multiple consumers can access the same APIs, all of them will benefit from the changes in those APIs. While changing, in order to not break the access of their consumers, certain parts of the APIs (i.e., signatures), including the name and parameter numbers of the APIs, cannot be altered. In such cases, e.g., it is unavoidable to change the signatures of APIs [27], the recommended approach is to introduce new APIs and encourage the consumers of the APIs to migrate to the new ones, meanwhile, the old APIs will be marked as deprecated.

API deprecation has become a common practice for evolving unwanted APIs, which are discouraged from using in new developments, and which will still be kept for a while (before removed) to preserve “backward compatibility” for existing consumers of the APIs. There are many valid reasons to deprecate APIs: the APIs are insecure, buggy, highly inefficient, or encourages poor coding practices, or will not be maintained anymore [10]. Taking these reasons and the natural progression of APIs in mind, our fellow researchers in the software engineering community have proposed various approaches to characterize the impact of API deprecation [13, 23, 33, 43–46, 50].

Unfortunately, despite that the literature has proposed numerous approaches targeting various aspects of deprecated APIs, covering many programming languages such as Java [22], Android [23], C# [7], etc., the deprecation of Python library APIs have never been the main focus. It is even more astonishing concerning that Python is the most popular programming language, according to the rank enumerated by IEEE Spectrum [12]. IEEE Spectrum attributes Python’s success to its explosion of new users in recent years, driven by easy-to-use yet capable Python libraries such as NumPy, Pandas, etc. Indeed, Python third-party libraries (hereafter, Python libraries) play an essential yet critical role in supporting the implementation of Python applications. APIs in such Python libraries, just like any other programming language, can also be deprecated. Indeed, as manifested in our empirical observation on Stack Overflow discussions, API deprecation in Python libraries has been continuously posted by developers, and the number of such discussions is even continuously increasing. This evidence shows that Python developers concern about API deprecation just as much as developers of other major programming languages. Therefore, the deprecation of Python library APIs should not be neglected by...
the community. There is a strong need to understand the status quo of deprecated Python APIs.

In this work, we attempt to fill the gap by presenting to the community the first exploratory study on the deprecation of APIs in popular Python libraries. We manually look into the implementation of six Python libraries that are among the top-10 popular libraries (based on the study of Pimentel et al. [38]) and that in total have been used by over 70,000 Python projects hosted on Github. We first aim at manually understanding how Python library contributors deprecate their APIs. Based on the manual observation, we further summarize the deprecated APIs and check against the official library documentation to examine if the deprecated library APIs are properly documented. If so, we go one step further to check if replacement messages are given in the documentation of deprecated APIs. Finally, we perform a large-scale study on the reaction of Python developers to deprecated library APIs. We select 1,200 Python projects on Github (Top-200 projects per library) to search for the usage of deprecated APIs and possible historical fixes of deprecated library APIs. We achieve this by introducing to the community a prototype tool called DLocator to automatically characterize the usage of deprecated APIs in Python repositories. The core contribution of DLocator is to statically resolve the challenges brought by the flexibility of Python’s API accessing mechanism (i.e., Python developers often map the long fully qualified API names to shorter ones), which has been deemed as a long-run challenge by the literature [34, 60].

Overall, our investigation into the deprecated Python APIs seeks to answer the following research questions (RQs):

- **RQ1: How do popular Python libraries deprecate their APIs?** In this very first research question, by investigating the deprecation strategies of popular Python libraries, we aim at understanding the current practices adopted by Python developers, in the hope of observing the root causes bearing the continuous increase of Python deprecation-related discussions on a developer-oriented online discussion website.

  **Main Findings:** Despite there is a specific package to deprecate Python library APIs, Python library contributors often ignore the recommended solution and implement ad-hoc strategies to deprecate APIs. These strategies are further different from libraries to libraries, and each library may adopt several strategies at the same time (yet in an inconsistent manner) to deprecate their no longer supported APIs.

- **RQ2: How are deprecated APIs documented in the Python Libraries?** The fact that Python libraries adopt ad-hoc strategies to deprecate APIs motivates us to investigate further how are the deprecated APIs documented. The ad-hoc implementations for deprecating Python APIs make it hard to invent automated approaches for managing the deprecated APIs, including their documentations. As a result, library contributors may have to rely on ad-hoc solutions as well to document the deprecated APIs, which, however, may be prone to mistakes.

  **Main Findings:** There are a significant number of deprecated library APIs that are not mentioned in the official library documentation. For the documented ones, unfortunately, not all of them have been provided with explicitly replacement messages for helping library users to avoid the usage of deprecated APIs.

- **RQ3: How are deprecated library APIs used by Python projects?** The findings of the previous two research questions further motivate us to go one step deeper to examine the impact brought by the fragmented implementation of API deprecation strategies and poor documentation of deprecated APIs. To this end, we are interested in knowing the Python developers’ reactions to API deprecation when maintaining their Python projects involving libraries with deprecated APIs.

  **Main Findings:** Python projects indeed use deprecated library APIs, for which the usage is positively correlated with the number of total accessed library APIs. Unfortunately, the usages of deprecated library APIs are rarely changed during the evolution of Python projects.

2 MOTIVATION

Python is an interpreted, high-level, general-purpose programming language [40]. Since 2003, Python has become one of the top-10 most popular programming languages listed by TIOBE Index [51]. From January 2020, it is listed as the third most popular language, following Java and C. Furthermore, Python won the “Programming Language of the Year” award [51] in 2007, 2010, and 2018, respectively. Many organizations, including Google, CERN, NASA, are using Python to build their rich software platforms and applications [41, 49, 57]. In the Python Package Index (PyPI) repository [39], there are more than 210,000 Python projects released with over 1,600,000 versions as of January 2020, which can assist developers in addressing a broad range of tasks.

The more libraries being leveraged by Python developers, the greater influence the deprecation of Python library APIs may lead to. Hence, there is a need to understand and quantify such influences. To better motivate this necessity of this, we now present a preliminary study on the extent of developer discussions related to the deprecation of Python library APIs.

2.1 Deprecation in Developer Discussions

As a preliminary study in understanding to what extent are deprecation problems discussed by software developers, we resort to public posts provided by developers on the famous developer Questions & Answer (Q&A) website, Stack Overflow[35]. We perform a search using composite conjunctions of keywords, including programming languages (i.e., Java, C, Python, C++, and C# [51]) and a category of deprecation-related keywords (i.e., deprecate, deprecated, and deprecation) with Sotorrent dataset, an official data dump of Stack Overflow [4]. We limit the search space into the questions that were posted from January 1st, 2009, to December 31st of 2018. In total, 1,063,837 posts are identified in this search.

We first assess how frequently the deprecation of each programming language is discussed on Stack Overflow by the numbers of Q&A posts that contain the deprecation-related keywords, of which results are presented in Figure 1. We observe that deprecation-related Java programs present is more discussed than that of the other four programming languages. Nevertheless, when reaching a peak in 2015, the number of discussions started to decrease. As can be seen from the figure, only the deprecation discussions related to Python are constantly increased over the period. This empirical
evidence shows that Python’s deprecation problems have attracted more and more attention from Python developers in recent years.

![Figure 1: Distribution of deprecation-related Q&A for the top-5 popular programming languages in the last decade.](image1)

Figure 1: Distribution of deprecation-related Q&A for the top-5 popular programming languages in the last decade.

### 2.2 Developers’ Sentiment Reaction to Deprecation

We then investigate the developers’ emotional attitude towards the discussions with respect to the deprecation, relying on Senti4D, a tool for analyzing sentimental polarity from software development corpus [8]. We apply Senti4D to all the deprecation-related posts we have identified previously. Developers’ sentiment reacted to deprecation problems will be grouped into three categories (i.e., negative, neutral, or positive). The experimental results are illustrated in Figure 2. We observe that Python has the lowest positive discussions about its deprecation problem compared with the other four programming languages. Additionally, for the negative sentiment, the related discussions for Python are close to or higher than other languages. It implies that Python developers are not happy with the current deprecation mechanism provided by or largely implemented in the Python community. Indeed, Figure 3 illustrates such an example (simplified from a Stack Overflow page), where a developer complains about the deprecation of a library API is not properly documented.

![Figure 3: Complaining the deprecation in Python library excerpted from Q&A on Stack Overflow.](image3)

Figure 3: Complaining the deprecation in Python library excerpted from Q&A on Stack Overflow.

Python becomes popular in the software community, and Python-related issues such as problems related to the deprecation of Python APIs are increasingly discussed on the online Question & Answer site. Like the reactions of other popular programming languages, Python developers lean to be negative as well when posting deprecation-related issues, Therefore, there is a need to empirically understand the status quo of deprecation in the Python realm, so as to recommend actionable steps to help developers better handle deprecation-related issues.

### 3 STUDY DESIGN

We now present the design details of our study, specifically, including the study subjects and the methodology of identifying deprecated APIs in Python codebase.

#### 3.1 Subject Selection

In this work, we are interested in understanding the deprecation of Python library APIs. As one of the most popular programming languages nowadays, Python has attracted a lot of attention from software developers who have already developed plenty of Python projects, including a large number of reusable packages (i.e., libraries). Indeed, there are at least 200,000 Python libraries available in the Python community, which are readily accessible for developers to choose from. It is nonetheless challenging to investigate all of the available libraries. Hence, we decide to leverage the famous Python libraries to form our study. In a recent study, Pimentel et al. [38] have reported the top-10 leveraged Python libraries, concerning their imported frequencies, in their large-scale empirical study. Among the ten libraries, four of them are provided as Python’s built-in modules, which should not be considered as third-party libraries. Therefore, we focus on our study on the remaining six Python libraries.

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Version</th>
<th>kLoc</th>
<th># Commits</th>
<th># Stars</th>
<th># Forks</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>1.8.1</td>
<td>432</td>
<td>22,038</td>
<td>12.9k</td>
<td>4.3k</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>3.2.0</td>
<td>857</td>
<td>32,418</td>
<td>10.8k</td>
<td>4.8k</td>
</tr>
<tr>
<td>Pandas</td>
<td>1.0.0</td>
<td>919</td>
<td>21,452</td>
<td>6.8k</td>
<td>3.1k</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>0.22.1</td>
<td>768</td>
<td>25,004</td>
<td>39.3k</td>
<td>19.2k</td>
</tr>
<tr>
<td>SciPy</td>
<td>1.4.1</td>
<td>729</td>
<td>22,498</td>
<td>6.8k</td>
<td>3.1k</td>
</tr>
<tr>
<td>Seaborn</td>
<td>0.9.0</td>
<td>80</td>
<td>2,450</td>
<td>6.8k</td>
<td>1.2k</td>
</tr>
</tbody>
</table>

Table 1 enumerates the selected six Python libraries, which are briefly described below:

- Numpy is a fundamental package for scientific computing [53].

We now briefly summarize the methodology we adopt in order to answer the first two research questions. Initially, we plan to write Python scripts to characterize the depreciation of Python library APIs automatically. However, as we will detail in the next section, Python library contributors do not follow the same strategy to deprecate APIs or provide means to programmatically document the deprecated APIs, making it hard to automatically achieve the purpose (i.e., identifying deprecated APIs and their documentation). Therefore, we resort to a manual approach to answer the first two research questions.

Regarding the third research question, which concerns the developers’ reaction to deprecated Python APIs, one of the core tasks towards answering this question is to check if a given deprecated API is used in a Python project. To achieve this purpose, we design and implement a prototype tool called DLocator to automatically characterize the usage of library APIs in Python projects. As acknowledged by Zhang et al. [60] in their recent study about the evolution of Python framework APIs, it is non-trivial to achieve such a purpose. Indeed, as justified by them, when accessing into Python library APIs, developers often map the long fully qualified API names to shorter ones (e.g., from `sklearn.feature_extraction.text.TfidfVectorizer` to `TfidfVectorizer`). Furthermore, APIs can be shortened in different methods (such as the four different API calling types shown in Figure 4) depending on developers’ programming behavior. Subsequently, such usages of shortened API versions cannot be statically reflected back to their fully qualified versions and can impede the identification of the same APIs with different usages. To resolve this challenge, we formatted the API calls presented in Python projects before characterizing them.

**Formatting API Calls.** We propose to format all API calls into the fully qualified name style that generally present APIs in a Python library in the form of `A.B.C.API_Name` (such as `sklearn.feature_extraction.text.TfidfVectorizer()`), where each identifier is separated with “.”, which represents the hierarchical structure of the full API path, including the location where the API is declared and the relationship between a class and its members.

The process of formatting API calls is illustrated in Algorithm 1. Given a Python program, we first parse it into an abstract syntax tree (AST) to facilitate traversing each entity of the program. We first collect the full paths (including the specific module names) for all API calls. If an AST node in the entire AST of the Python program is an import statement node [1], its information related to API will be extracted. If the import statement node has an alias, the alias (such as 3rd one in Figure 4) is extracted as the retrieving key for API mapping (cf. Line-9). Otherwise, the import item is collected as the retrieving key (cf. Line-11). If the import statement node represents an import statement, the full module name is identified as the path value for API mapping (cf. Line-13). While if the import statement node is a from-import statement, the full module name concatenated with the import item is identified as the related value for API mapping (cf. Line-15). We then collect and format all API calls from the Python program code. For each AST node in $P_{AST}$, if it is a function call node, the related function name is collected as an API call candidate (cf. Line-18). After checking all AST nodes, each API call candidate is validated whether it is invoked as an API call from a library. If so, the related full qualified name is concatenated to format each API call.

**4 STUDY RESULTS**

We now provide the experimental results along with the key insights observed for answering aforementioned research questions.

**4.1 RQ1: Declaration of Deprecated APIs**

Our first research question concerns the declaration of deprecated APIs. We want to understand how Python library contributors deprecate their library APIs. We hence manually look into the open-source repositories of the selected six libraries.
API cases are caused by the parameter’s value. Nevertheless, the Table 2 illustrates the distributions of API deprecations at each Java), Python API deprecations present a finer granularity. Except maintaining tasks for developers. deprecated APIs with respect to parameters raise a challenge in parameter’s value levels.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric_only</td>
<td>raw</td>
<td>reduced</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>30</td>
<td>58</td>
<td>25</td>
</tr>
</tbody>
</table>

We further investigate how the deprecated APIs are declared in Python programs by developers. According to our investigation, the API deprecations are generally declared in three strategies:

1. **Decorator**: The strategy used by library contributors to declare the deprecation of Python APIs is through the so-called Python decorator mechanism. Similar to the Java annotation mechanism, Python decorator provides a means to extend the behavior of a function without explicitly modifying it. Figure 6 presents a concrete example showing how the Python decorator is used to deprecate APIs in class LogTransformBase (e.g., line 3). Unfortunately, different library contributors tend to maintain their own decorator implementations instead of adopting a common one such as the deprecation project hosted on PyPI, the official third-party software repository for Python. This ad-hoc decorator implementation makes it hard to invent generic scripts to automatically characterize deprecated APIs for Python libraries.

2. **Hard-coded warning**: The declaration of API deprecation is realized by providing hard-coded warning messages to API users at runtime. Figure 6 demonstrates such an example, where the deprecated API \texttt{GradientBoostingClassifier()}\footnote{one of its parameters is deprecated} is highlighted by a hard-coded warning message (line 10).

3. **Comments**: The deprecation is declared as comments (i.e., natural language) in the source code. Let us take Figure 6 again as an example, in line 20, the comments of API \texttt{sleipn\textunderscore} explicitly note readers that the API is deprecated in SciPy 1.1.

Table 3 summarizes the declaring strategies of deprecated APIs adopted by the selected popular Python libraries. It is interesting to observe that: (1) Generally, most of the libraries (except Seaborn) leverage Python decorator to declaring the deprecation of APIs. Similar methodologies have been adopted by Java as well (via \texttt{@Deprecated} annotation), which has been demonstrated to

Our observation reveals that Python APIs are deprecated at different granularities, which are summarized into four categories: function, parameter, parameter’s value, and others.

- **Function**: A function/class defined in a library is simply no longer supported by the library, such as the three examples shown in Figure 6. The deprecated Python class is grouped into this category as the invocation of Python class is similar to the function call (like constructor calls in Java).

- **Parameter**: The positional or keyword argument [14] defined in an API can be deprecated in Python. For example, parameter "numeric-only" of function pandas.\texttt{DataFrame.min}\footnote{will no longer be supported (lines 10-11 in Figure 5)} is deprecated and will be replaced with "skipna".

- **Parameter’s Value**: This type refers to cases where an argument of an API will no longer accept certain value(s) as input. For example, values “full” and “economic” for argument mode of API numpy.\texttt{linalg.qr}\footnote{will no longer be supported (lines 10-11 in Figure 5)}.

Comparing with other popular programming languages (e.g., Java), Python API deprecations present a finer granularity. Except the code entities (i.e., classes, and functions) that can be deprecated by developers in different programming language code, the parameter and its value could be discarded by Python contributors as well. Table 2 illustrates the distributions of API deprecations at each of the three granularities. Overall, functions are the main entities that will be deprecated by Python contributors, it is similar to the deprecated entities in other programming languages. In Pandas, Scikit-learn, and SciPy, the deprecated parameters hold a significant share of API deprecations. Only a small number of deprecated API cases are caused by the parameter’s value. Nevertheless, the deprecated APIs with respect to parameters raise a challenge in maintaining tasks for developers.

### Algorithm 1: Formatting API Calls.

```python
1. Function format (P):
2. A ← importMap () /* Initiate a map for API calls. */
3. As ← importMap () /* Initiate a list for formatted API calls. */
4. C ← importList () /* Initiate a list for function calls. */
5. P_AST ← parsePythonCode (P) /* Parse the Python code into an AST. */
6. /* P_AST: an AST node in P_AST. */
7. foreach N_AST in P_AST do
8. if isImportStatement (N_AST) then
9. if hasAlias (key) then
10. key ← getAlias (N_AST)
11. else
12. key ← getModuleName (N_AST)
13. if isFromImportStatement (N_AST) then
14. value ← getModuleName (N_AST)
15. else if isFromImportStatement (N_AST) then
16. value ← getModuleName (N_AST from N_AST import)
17. dict[key, value] = value
18. else if isFunctionCallNode (N_AST) then
19. C.addFunctionCall (N_AST)
20. end if
21. endforeach
22. C/fullQualifiedModuleName ← retrieve/fullQualifiedModuleName (A)
23. if fullQualifiedModuleName ≠ null then
24. formattedAPICall ← format (c, fullQualifiedModuleName)
25. As.add (formattedAPICall)
26. return As
```

Our observation reveals that Python APIs are deprecated at different granularities, which are summarized into four categories:

- **Function**: A function/class defined in a library is simply no longer supported by the library, such as the three examples shown in Figure 6. The deprecated Python class is grouped into this category as the invocation of Python class is similar to the function call (like constructor calls in Java).

- **Parameter**: The positional or keyword argument [14] defined in an API can be deprecated in Python. For example, parameter "numeric-only" of function pandas.\texttt{DataFrame.min}\footnote{will no longer be supported (lines 10-11 in Figure 5)} is deprecated and will be replaced with "skipna".

- **Parameter’s Value**: This type refers to cases where an argument of an API will no longer accept certain value(s) as input. For example, values “full” and “economic” for argument mode of API numpy.\texttt{linalg.qr}\footnote{will no longer be supported (lines 10-11 in Figure 5)}.

Comparing with other popular programming languages (e.g., Java), Python API deprecations present a finer granularity. Except the code entities (i.e., classes, and functions) that can be deprecated by developers in different programming language code, the parameter and its value could be discarded by Python contributors as well. Table 2 illustrates the distributions of API deprecations at each of the three granularities. Overall, functions are the main entities that will be deprecated by Python contributors, it is similar to the deprecated entities in other programming languages. In Pandas, Scikit-learn, and SciPy, the deprecated parameters hold a significant share of API deprecations. Only a small number of deprecated API cases are caused by the parameter’s value. Nevertheless, the deprecated APIs with respect to parameters raise a challenge in maintaining tasks for developers.

### Table 2: Distributions of deprecated APIs in each category.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Function</th>
<th>Parameter</th>
<th>Parameter’s Value</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>26</td>
<td>2</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>47</td>
<td>7</td>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>Pandas</td>
<td>14</td>
<td>11</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>19</td>
<td>18</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>SciPy</td>
<td>23</td>
<td>17</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>Seaborn</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>135</td>
<td>56</td>
<td>13</td>
<td>204</td>
</tr>
</tbody>
</table>

We further investigate how the deprecated APIs are declared in Python programs by developers. According to our investigation, the API deprecations are generally declared in three strategies:

1. **Decorator**: The strategy used by library contributors to declare the deprecation of Python APIs is through the so-called Python decorator mechanism. Similar to the Java annotation mechanism, Python decorator provides a means to extend the behavior of a function without explicitly modifying it. Figure 6 presents a concrete example showing how the Python decorator is used to deprecate APIs in class LogTransformBase (e.g., line 3). Unfortunately, different library contributors tend to maintain their own decorator implementations instead of adopting a common one such as the deprecation project hosted on PyPI, the official third-party software repository for Python. This ad-hoc decorator implementation makes it hard to invent generic scripts to automatically characterize deprecated APIs for Python libraries.

2. **Hard-coded warning**: The declaration of API deprecation is realized by providing hard-coded warning messages to API users at runtime. Figure 6 demonstrates such an example, where the deprecated API \texttt{GradientBoostingClassifier()}\footnote{one of its parameters is deprecated} is highlighted by a hard-coded warning message (line 10).

3. **Comments**: The deprecation is declared as comments (i.e., natural language) in the source code. Let us take Figure 6 again as an example, in line 20, the comments of API \texttt{sleipn\textunderscore} explicitly note readers that the API is deprecated in SciPy 1.1.

Table 3 summarizes the declaring strategies of deprecated APIs adopted by the selected popular Python libraries. It is interesting to observe that: (1) Generally, most of the libraries (except Seaborn) leverage Python decorator to declaring the deprecation of APIs. Similar methodologies have been adopted by Java as well (via \texttt{@Deprecated} annotation), which has been demonstrated to
Table 3: The declaring strategies of deprecated APIs in six Python libraries.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Decorator</th>
<th>Hard-code warning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>np.decorator</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>mpl.decorator</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pandas</td>
<td>pd.decorator</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>sk.decorator</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SciPy</td>
<td>np.decorator</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Seaborn</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

"np", "pd", "mp" and "sk" represent NumPy, Matplotlib, Pandas and Scikit-learn that define the decorators.

be effective. Unfortunately, unlike what occurs in Java, where all the projects use the same mechanism to manage deprecated APIs, Python library contributors maintain their depiction decorator that is different from the ones adopted by other libraries (except for SciPy, which leverages the decorator supported by NumPy). (2) All six libraries attempt to describe API deprecation via developer comments. However, when commenting on the deprecation of APIs, Python libraries do not provide structural mechanisms (such as @deprecated Javadoc) to help in maintaining the up-to-date documentation of deprecated APIs, resulting in difficulties to propagate the deprecation to the API users. (3) To mitigate this problem, all the libraries have decided to warn library users at runtime about their usages of deprecated APIs.

Among the six selected libraries, four of them have leveraged all the three strategies to declare the deprecation of APIs. These three strategies seem to complement each other: Decorator allows maintainers to programmatically manipulate the deprecated APIs but does not provide a detail explanation on why is the API deprecated. Comments can then be leveraged to fill the gap by providing natural language explanation to library developers. Finally, the hard-coded warning provides an effective means to further propagate the deprecation message to library users. To check if library contributors are actually using different strategies to complement each other when deprecating APIs, we are further motivated to answer the following question: To what extent are the identified strategies adopted to declare deprecated APIs?

Table 4 summarizes our empirical findings. The majority of deprecated APIs, in all the selected libraries, is declared by a single strategy, as shown in columns 2-4. The remaining deprecated APIs are mostly declared by two strategies: A large part of them are deprecated by hard-coded warnings plus comments (column 5), while a small number of them by decorator plus comments (column 6). There are no deprecated APIs that are declared by both decorator and hard-coded warning strategies (column 7 and column 8). This evidence suggests that Python library contributors have poorly handled API deprecation in the community. Although different strategies are leveraged, they do not take efforts to ensure the consistency among them, missing the opportunities to complementarily and hence thoroughly declare the deprecation of APIs. This problem also explains why the number of Python deprecation discussions on developer Q&A site continuously increases, as disclosed in Section 2.

Table 4: The number of deprecation occurrences in the subject dataset by different ways

<table>
<thead>
<tr>
<th>Subject</th>
<th>C</th>
<th>W</th>
<th>D</th>
<th>C+W</th>
<th>D+W</th>
<th>C&gt;D</th>
<th>D&gt;C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>5</td>
<td>8</td>
<td>42</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>Pandas</td>
<td>1</td>
<td>13</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>10</td>
<td>2</td>
<td>15</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>SciPy</td>
<td>12</td>
<td>5</td>
<td>14</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>43</td>
</tr>
<tr>
<td>Seaborn</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>32</td>
<td>37</td>
<td>81</td>
<td>48</td>
<td>6</td>
<td>0</td>
<td>204</td>
<td></td>
</tr>
</tbody>
</table>

*C, W, D* represent comments, hard-coded warning and decorator declaring strategies, respectively.

The deprecation of Python library APIs is mainly declared via three strategies: Decorator, Hard-coded warning, and comments. Library contributors, nevertheless, do not follow the same paradigm but resort to different strategies, which are often customized and maintained by their contributors, to deprecate APIs. Moreover, there are cases that deprecated APIs may be declared by two strategies. Nonetheless, there is no single API that is deprecated via all the three identified complementary strategies, i.e., lacking consistency between the adopted deprecation strategies.

4.2 RQ2: Documentation of Deprecated APIs

For the future evolution and maintenance of programs, it is worthy of providing clear documentation for deprecated APIs [23, 44, 62]. Indeed, as stated in the literature [13], it is essential to provide sufficient information concerning deprecated APIs for library users. Mainly, documenting APIs is known as one of the most effective methods to resolve deprecated APIs since they can convey the intentions of why APIs are deprecated [36].

Therefore, in the second research question, we concern about the documentation of deprecated Python library APIs. The first task is to identify the artifacts that document the deprecation of library APIs. In this work, we resort to the official documentation (also known as API reference) and release notes to check if the
deprecation messages are timely conveyed to library users. If so, we consider the API deprecation is well documented; otherwise, we regard the deprecation of APIs as undocumented ones.

Table 5 summarizes the number of undocumented deprecated APIs. There are in total of 49 deprecated APIs (among the six selected popular libraries) that are not documented by developers. Specifically, concerning the three deprecation strategies, i.e., comments, hard-coded warning, and decorator, the numbers of undocumented APIs are 12 (24%), 14 (28%), and 29 (33%), respectively. Concerning each library along, to the best, only 8% of deprecated Pandas APIs are not documented; to the worst, the undocumented rate of deprecated SciPy APIs reaches as high as 47%.

Table 5: The number of undocumented/total deprecated APIs in the six subjects.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Comments</th>
<th>Warning</th>
<th>Decorator</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>2/20</td>
<td>3/25</td>
<td>4/4</td>
<td>7/30  (23%)</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>2/8</td>
<td>3/11</td>
<td>1/42</td>
<td>6/58  (10%)</td>
</tr>
<tr>
<td>Pandas</td>
<td>1/6</td>
<td>1/17</td>
<td>0/6</td>
<td>2/25  (8%)</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>1/24</td>
<td>1/14</td>
<td>12/17</td>
<td>13/41 (32%)</td>
</tr>
<tr>
<td>SciPy</td>
<td>6/24</td>
<td>5/13</td>
<td>12/18</td>
<td>20/43 (47%)</td>
</tr>
<tr>
<td>Seaborn</td>
<td>0/4</td>
<td>1/4</td>
<td>0/0</td>
<td>1/7   (14%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>12/36(14%)</strong></td>
<td><strong>14/84(16%)</strong></td>
<td><strong>29/87(33%)</strong></td>
<td><strong>49/204(24%)</strong></td>
</tr>
</tbody>
</table>

*6 undocumented API deprecations are declared with both comments and warning strategies, or both comments and decorator strategies, thus the number in column “Total” is slightly lower than the sum of the previous related three columns. The same as Table 6.*

We further investigate whether the alternatives to deprecated APIs are provided when their documents are available, of which results are summarized in Table 6. The correctness of the results were confirmed by two PhD students who have Python as their primary programming language for daily tasks but have been unaware of the deprecation study for Python. These two PhD students are invited to independently verify all of the 155 documented deprecated APIs, among which they have agreed on 153 of them after reading documentation text of these APIs, giving a substantial inter-rater agreement as suggested by Cohen’s kappa coefficient (kappa=0.663). The disagreements are mainly related to semantic differences of the descriptive language mentioning of these APIs (without explicitly mentioning the deprecation or removal of the API).

Overall, around one-third of documented API deprecations do not provide alternatives for users, which is in line with the results reported by Brito et al. [7] in a contemporary study for the deprecated APIs in Java programs. We observe that, 41% of deprecated APIs declared with decorator are not provided with the alternatives, that is much higher than comments and hard-coded warning. Furthermore, in 46 deprecated APIs that are not provided with replacements, half (24) of them are declared with the decorator that is followed by the comments and hard-coded warning strategies.

While going through the documents of deprecated APIs, we further observe that some of such documents are un-clearly, vaguely or even incorrectly described. For example, Figure 7 presents an example of unclear documentation of deprecated APIs, where parameter min_impurity_split is deprecated in “all the tree-based estimators are deprecated” that actually refers to 7 deprecated APIs in the context. However, the 7 APIs are not clearly described in this document. Developers will take more efforts to figure out what are the all tree-based estimators for their programming tasks. Without giving explicit and definitive information, it is inconvenient for developers to avoid the usage of those deprecated library APIs. In Figure 8, the deprecated API NaturalLogTransform is incorrectly presented as NaturalLogTransformLog in Figure 8, which can confuse even mislead the library users.

Table 6: The number of deprecated APIs that are documented but not provided with replacement messages.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Comments</th>
<th>Warning</th>
<th>Decorator</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>4/18</td>
<td>4/22</td>
<td>0/0</td>
<td>4/23  (17%)</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>1/6</td>
<td>0/8</td>
<td>18/41</td>
<td>19/52 (37%)</td>
</tr>
<tr>
<td>Pandas</td>
<td>2/5</td>
<td>3/16</td>
<td>0/6</td>
<td>3/23  (13%)</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>7/23</td>
<td>3/13</td>
<td>5/5</td>
<td>11/28 (39%)</td>
</tr>
<tr>
<td>SciPy</td>
<td>6/18</td>
<td>3/8</td>
<td>1/6</td>
<td>6/23  (27%)</td>
</tr>
<tr>
<td>Seaborn</td>
<td>2/4</td>
<td>2/3</td>
<td>0/0</td>
<td>3/6   (30%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>22/74(30%)</strong></td>
<td><strong>15/70(21%)</strong></td>
<td><strong>24/58(41%)</strong></td>
<td><strong>46/155(30%)</strong></td>
</tr>
</tbody>
</table>

### API changes summary

**Trees and ensembles**

- Gradient boosting base models are no longer estimators. By Andreas Müller.
- All tree based estimators now accept a min_impurity_decrease parameter in lieu of the min_impurity_split, which is now deprecated. The min_impurity_decrease helps stop splitting the nodes in which the weighted impurity decrease from splitting is no longer at least min_impurity_decrease. #8449 by Raghav RV.

**Figure 7:** Example of unclear presentation of a deprecated API excerpted from Scikit-learn’s documents.

**Figure 8:** Example of incorrect document of a deprecated API excerpted from Matplotlib’s documents: the deprecated API NaturalLogTransform is incorrectly presented as NaturalLogTransformLog.

Around one-quarter of deprecated APIs are not mentioned in the library’s official documentation, making it hard for library users to mitigate the usage of deprecated APIs. For such deprecated APIs that are documented, 70% of them further come up with alternatives, which is in line with other major programming languages.

---


4. https://matplotlib.org/3.1.0/api/prev_api_changes/api_changes_3.1.0.html#transforms-scales
4.3 RQ3: Usages of Deprecated APIs

The findings we have observed for answering the previous two research questions show that deprecated APIs are declared in a disordered manner (multiple, non-generic ad-hoc strategies) and yet are also not well documented. Hence, we hypothesize that this disorganization may be propagated to Python library users. To this end, in the last research question, we are interested in exploring how are these deprecated APIs used in real Python projects and to what extent do developers address them.

A Dataset of GitHub Repositories. To support the experiments, we resort to GitHub to collect open-source Python project clients of each subject selected in this work. In particular, we crawl the top-200 best-match-ranked projects returned by the GitHub search, with the composite searching condition of the subject library name and Python repositories only. Note that, the official repositories of the six subjects are excluded from the searching results. The search engine of GitHub is only a simple text-based tool that receives as input a list of query words, thus the searched results are not fully related to the searching condition [59]. To reduce the noise, the searched projects are further purified by checking whether they indeed invoke the API calls from the related subject library. Eventually, as presented in Table 8, a total of 916 Python projects that invoke 200k API calls from six Python libraries are collected as their client projects.

With the selected 916 repositories, we first collect commits with respect to “fixing deprecation” by matching commit messages with fix-related and deprecation-related keywords (i.e., fix and deprecated). For all the commits of selected Python projects, we observe that commit messages mentioning deprecation only account for a very small portion (less than 0.1%). Yet, the number of commit messages mentioning both keywords (indicating possible fixes of deprecated APIs) is even much smaller.

DLocator for Locating Deprecated APIs. Motivated by this finding, for which Python developers are rarely fixing deprecated library APIs from the commit messages point of view, we go one step deeper to directly look at the evolution of Python code, since commit messages may not necessarily reflect the exact changes of the code due to the diversity of human language (e.g., word choices, or presentation styles) for conveying the knowledge. To this end, we propose to the community a prototype tool called DLocator5, which takes as input a Python Git repository and a set of deprecated APIs and outputs the set of deprecated APIs that are currently used by the project. To resolve the challenges demonstrated in Figure 4, DLocator implements the API call formatting algorithm (as shown in Algorithm 1) as one of the core contributions to locate the usage of APIs. By looking at the evolution of the project (e.g., visiting all the commits), DLocator can further spot all the historical deprecated APIs that are no longer presented at the latest version of the project, and are hence regarded as fixed ones.

Table 7 summarizes the usages of deprecated APIs in the latest version of all client Python projects, where the “Threshold-X” columns mean that the client project using the deprecated APIs for at least X times is considered. “# Projects” columns list the number of projects that are using the deprecated any API calls. For example, the cell (8/181 in column-3 and row-3) shows that in 181 client projects of NumPy library, 8 of them are using the deprecated APIs for at least one time in their latest version. Columns “# deprecated API calls” list the number of distinct/all deprecated API calls used in the client projects. For example, the cell (55%=93/169) by its client projects, which is followed by Scikit-learn, Pandas, SciPy, Seaborn. From the aspect of depth that the times of used deprecated APIs, the main merchants of deprecated APIs swift to Pandas, Scikit-learn, and Seaborn. From the aspect of spread that the number of client projects using deprecated APIs, the deprecated APIs in Pandas are most widely (55%=93/169) by its client projects, which is followed by Scikit-learn, and Seaborn. As shown in Figure 9, the majority of client projects do not remove deprecated APIs in their evolution process, which suggests that Python developers do not pay enough efforts to address the deprecated API calls.

We observe that overall, 119 out of 717 deprecated APIs of six libraries are still used in 18%(=169/916) client projects for 1,584 times. From the aspect of spread that the number of client projects using deprecated APIs, the deprecated APIs of Pandas are most widely (55%=93/169) by its client projects, which is followed by Scikit-learn, and Seaborn. From the aspect of depth that the times of used deprecated APIs, the deprecated APIs in Pandas are most (40%=638/1584) recurrently used in client projects, which is followed by Scikit-learn and SciPy. While from the aspect of the number of distinct used deprecated APIs, the main merchants of deprecated APIs swift to SciPy, Scikit-learn and Matplotlib. When increasing the threshold to 10, 20 and 50, deprecated API usages present the same distributions as the Threshold-1. These results indicate that the recurrent usages of deprecated APIs mainly focus on some specific API deprecations.

Figure 9 presents the distribution of the 169 client projects with respect to changing deprecated API calls, where “Decreased” represents that the number of deprecated APIs called in the present versions of all client projects is decreased comparing with the deprecated APIs invoked in the whole evolution histories of all client projects. “Non-decreased” denotes that opposite situation. As shown in Figure 9, the majority of client projects do not remove any deprecated APIs during their evolution process, which suggests that Python developers do not pay enough efforts to address the deprecated API calls.

Figure 10 further illustrated the times of the deprecated APIs invoked by the client projects in the present time (i.e., “Present”

5Available at https://github.com/dlocator-dev/dlocator

Figure 9: Distribution of projects with respect to the evolution of deprecated API calls.

Figure 10: Distribution of deprecated APIs that are presently/historically used by the client projects.
Table 7: The usage of deprecated APIs in the latest version of selected client projects.

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Threshold-1</th>
<th>Threshold-10</th>
<th>Threshold-20</th>
<th>Threshold-50</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>36</td>
<td>8/181</td>
<td>5/20</td>
<td>8/140</td>
</tr>
<tr>
<td>Pandas</td>
<td>145</td>
<td>93/167</td>
<td>17/638</td>
<td>75/107</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>130</td>
<td>31/162</td>
<td>34/456</td>
<td>30/121</td>
</tr>
<tr>
<td>SciPy</td>
<td>101</td>
<td>9/100</td>
<td>35/305</td>
<td>8/35</td>
</tr>
<tr>
<td>Seaborn</td>
<td>15</td>
<td>23/135</td>
<td>4/57</td>
<td>12/40</td>
</tr>
<tr>
<td>Total</td>
<td>717</td>
<td>169/916</td>
<td>119/1,584</td>
<td>138/584</td>
</tr>
</tbody>
</table>

Table 8: Overview of selected datasets. Only such projects that have accessed at least one library API are considered.

<table>
<thead>
<tr>
<th>Subject</th>
<th>NumPy</th>
<th>Matplotlib</th>
<th>Pandas</th>
<th>Scikit-learn</th>
<th>SciPy</th>
<th>Seaborn</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawled Projects</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>1,200</td>
</tr>
<tr>
<td>Considered Projects</td>
<td>181</td>
<td>171</td>
<td>167</td>
<td>162</td>
<td>100</td>
<td>135</td>
<td>916</td>
</tr>
<tr>
<td>Total API calls</td>
<td>112,427</td>
<td>10,624</td>
<td>16,011</td>
<td>45,567</td>
<td>18,217</td>
<td>1,479</td>
<td>284,325</td>
</tr>
</tbody>
</table>

Table 9: The top-3 most recurrently used APIs in the present/t/historical versions of client projects.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Present</th>
<th>History</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>numpy.load</td>
<td>numpy.loads</td>
</tr>
<tr>
<td></td>
<td>numpy.array</td>
<td>numpy.asscalar</td>
</tr>
<tr>
<td></td>
<td>numpy.mirr</td>
<td>numpy.mirr</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>matplotlib.ticker logged</td>
<td>matplotlib.ticker.LoggedAxis</td>
</tr>
<tr>
<td></td>
<td>matplotlib.ticker.Logged</td>
<td>matplotlib.axis.Logged</td>
</tr>
</tbody>
</table>
| Pandas | pandas.ts_friendly | pandas.ts_friendly />
| | pandas.ts_friendly | pandas.ts_friendly />
| Scikit-learn | sklearn.linear_model.LogisticRegression | sklearn.linear_model.LogisticRegression />
| | .checkConsistency | .checkConsistency />
| Seaborn | seaborn.factorplot | seaborn.factorplot />
| | seaborn.despine | seaborn.despine />
| | seaborn.tsplot | seaborn.tsplot />

5 DISCUSSION

5.1 Implication

Based on our empirical findings, we now discuss possible implications that our community could build upon for appropriately handling deprecated Python library APIs.

Built-in Python Module to Handle API Deprecation. Overall, our investigations reveal that in popular Python libraries, deprecated APIs are generally declared through three categories: Decorator, Hard-coded warning, and Comments. Yet, the library contributors do not follow the same paradigm but resort to different strategies to deprecate APIs, which makes it more complicated to maintain the deprecated APIs in Python libraries than the deprecation of many other programming languages. To mitigate this problem, we argue that there is a strong need to invent a generic deprecation paradigm that is acceptable to all the Python developers and library contributors. Ideally, it would be great if such a paradigm could be provided as a built-in Python module and thereby contributing to the long-term health of third-party libraries in the Python ecosystem.

Enforce Consistency among the Complementary Deprecation Strategies. As discussed earlier, the three deprecation strategies currently leveraged by library contributors are complementary to each other. Indeed, the strategies provide different functions to the deprecation of Python APIs. Therefore, we suggest that Python library contributors should declare deprecated APIs with all the three strategies simultaneously and should make efforts to ensure consistency among the declarations.

Structured Rules to Generate and Maintain the Documentation of Python (Deprecated) APIs. The fact that Python library’s official documentation, including its API references, has undocumented a significant number of deprecated APIs, and some of the explanations for the documented APIs are vague and even incorrect, suggests that automated approaches are demanded to manage the documentation of (deprecated) Python APIs. In Java, developers use the Javadoc mechanism with structured keywords (such as @deprecated) to automatically generate and maintain the documentation of (deprecated) APIs, which has been demonstrated to be effective and useful. In C++, attribute ([[deprecated]]) can be leveraged to tag an entity as deprecated in the source code [9], which subsequently will allow the compilers to capture such deprecations. In C#, ObsoleteAttribute class can be placed to mark the program components that are no longer supported [11]. Therefore, in the realm of Python, we believe similar strategies should also be introduced to manage the documentation of Python APIs, including the deprecated ones. And similar to that of API deprecation, such
strategies should be generic and in the best scenario provided as a built-in Python module.

**Automated Toolchain to Pinpoint (and fix if possible) the usage of Deprecated APIs.** Last but not least, no matter how well the deprecated library APIs managed (e.g., with a clear methodology and support of automated toolchain), library users, in any case, will likely still access into deprecated APIs. Therefore, we believe that it is not only important to propose automated toolchains for helping library contributors better manage their deprecated APIs, but also essential to develop automated toolchain for helping Python developers avoid the usage of such deprecated APIs (e.g., providing real-time alerts when developers are programming with Python IDEs).

5.2 Threats to Validity

**Internal validity.** The major threat to the internal validity of our study lies in possible errors in the implementation of our experimental scripts and tools. To mitigate this threat, we have carefully reviewed the toolchains and manually validate partial experimental results against selected benchmarks. Furthermore, our static analyzer DLocator currently only locates deprecated APIs at the function, parameter level and will ignore such deprecation caused by parameter due to the lack support of constant propagation. Therefore, it will likely yield false positives results when locating deprecated APIs in open-source Python projects. We plan to mitigate this problem in our future work. Finally, as a significant amount of tasks that are conducted by the authors manually, yet human is prone to making mistakes, the experimental observations thereby may also contain mistakes. The authors have hence cross-evaluated the experimental results to mitigate this potential threat.

**External validity.** The primary threat to the external validity of our study concerns the choice of selected Python libraries and their client Python projects. In this work, we only consider third-party libraries and have only selected six libraries. For each library, we have only considered 200 Github projects. Therefore, all the findings might only be valid to those libraries and Python projects, and cannot be generic to all the other Python libraries and projects. Nevertheless, our experiments are conducted based on the most popular third-party libraries (In total, the six libraries are used by more than 70K Python projects, based on the records of Github) and top-ranked Github projects. The actual situation on how Python library APIs are deprecated may be even worse in the Python community. We plan to continue exploring towards this direction in our upcoming works.

6 RELATED WORK

As the evolution of software programs, the deprecation of APIs arises as an inevitable circumstance for developers and practitioners. Nevertheless, the deprecated APIs can have a huge impact on the ecosystem, either considered in terms of projects or developers that are impacted by the change, or measured by the overall amount of changes [43, 56]. In the literature, various studies on deprecated APIs for different ecosystems have been conducted to boost the momentum of characterizing API deprecations.

Robbes and his colleagues [43] reported an empirical study on the ripple effects as a result of API deprecations across an entire Smalltalk ecosystem, to investigate how developers react to the API deprecation. Espinha et al. [13] performed a semi-structured interview with six developers to understand the distress caused by the evolution of web APIs, and further explored how four web API providers organize their API evolution and the impact of web service API evolution on their clients. Sawant et al. [45, 46] investigated the effect of deprecations of five Java APIs on 25,357 clients. Later, they further conducted semi-structured interviews [44] with 17 third-party Java API producers and survey 170 Java developers to gain a deep understanding of the requirements regarding deprecation from API producers and consumers. More recently, they extracted the patterns of reaction to API deprecation to ascertain the scale of reactions or non-reactions of users to deprecated entities [47]. These work mainly focused on analyzing the impact of API changes [16, 17, 25, 26, 28], but they did not investigate how API contributors handle (i.e., declare and document) API deprecations explored in our work.

Ko et al. [22] pointed out that 61% of deprecated APIs are offered with replacements after examining 260 deprecated APIs from 8 Java libraries and their documentation. Brito et al. [6] also have a similar observation that 64% of deprecated APIs are provided with alternatives for users, after analyzing 661 real-world Java systems. According to the analysis of replacement messages of deprecated API elements in 622 Java and 229 C# systems, Brito et al. [7] found that 66.7% and 77.8% of the API elements are deprecated with replacement messages per project. From the aspect of replacement offers, the deprecated APIs in Java and C# programs outperform the deprecated APIs in Python programs studied in this work.

Raemaekers et al. [42] observed that developers do not follow deprecation patterns as suggested by semantic versioning, after conducting an investigation of deprecation patterns in 22,205 Java libraries. Zhou et al. [62] conducted a retrospective analysis of API deprecations in 26 open-source Java systems over 690 versions and reported that API deprecation messages are not well maintained by API producers and only a small part of deprecated APIs are specified with related replacements. In our study, we observe that the deprecated APIs in Python libraries are documented without a unique style as well, which could impede the identification of deprecated APIs for users.

Mirian et al. [31] presented the analysis of web feature deprecation through the lens of the Chrome browser and revealed six reasons behind the fact that developers would want to deprecate web features. Li et al. [23, 24] proposed to characterize the deprecated APIs in Android ecosystems with a systematic code mining of 10,000 Android applications. In this study, we focus on characterizing the API deprecations in Python third-party libraries, to the best of our knowledge, that is not explored in the literature. Besides characterizing the API deprecations, reactions with respect to API deprecation/evolution have been studied in various scenarios [54, 55]. Montandon et al. [32] studied a lesson-learned approach to documenting APIs with examples. Hou et al. [19] explored developers’ intent behind API evolution. Xavier et al. [58] analyzed the impact of API breaking changes with a large-scale study for historical changes of APIs. Various approaches (e.g., deep learning based [2, 3, 27], model based [5] and dictionary based [21]) have been studied in the literature to refactor the inaccurate API
names [18]. Henkel and Diwan [15] proposed a lightweight approach for capturing and replaying refactorings to support API evolution. Our work is to analyze the characteristics of deprecated APIs in Python libraries, which can build knowledge for the study scenarios similar to previous works.

7 CONCLUSION

We presented an exploratory study on how deprecated APIs are currently declared and documented by Python library contributors and how Python developers react to such API deprecations. To answer these research questions, we have manually dug into the implementation of six reputed Python libraries and empirically found that API deprecation in Python libraries is currently handled in chaos, i.e., declared with ad-hoc strategies and lacking proper documentation. By further looking into the usage of deprecated APIs in 1,200 popular Python projects, we experimentally reveal that Python developers have not paid much attention to the deprecation of library APIs. These findings strongly suggest that our community should take immediate actions to invent reliable approaches for helping both library contributors and users to appropriately handle the deprecation of library APIs.

ACKNOWLEDGEMENTS

This work is supported by the Project 1015-YAH20102, the National Science Foundation of China (Grant No. 61802180), the Natural Science Foundation of Jiangsu Province (Grant No. BK20180421), the National Cryptography Development Fund (Grant No. MMJ[20180105) and the Fundamental Research Funds for the Central Universities (Grant No. NE2018106).

REFERENCES
