Automatically Fixing Breaking Changes of Data Science Libraries

Hailie Mitchell
mitchelh@dickinson.edu
Dickinson College, Carnegie Mellon University
Carlisle, Pennsylvania, USA

ABSTRACT

Data science libraries are updated frequently, and new version releases commonly include breaking changes. These are updates that cause existing code to not compile or run. Developers often use older versions of libraries because it is challenging to update the source code of large projects. We propose CombyInferPy, a new tool to automatically analyze and fix breaking changes in library APIs. CombyInferPy infers rules from the history of library source code in the form of Comby templates, a structural code search and replace tool that can automatically transform code. Preliminary results indicate CombyInferPy can update the pandas library Python code. Using the Comby rules inferred by CombyInferPy, we can automatically fix several failing tests and warnings. This shows this approach is promising to help developers update libraries.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Software libraries and repositories**.

KEYWORDS

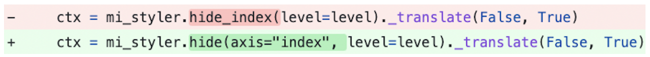
software maintenance, machine learning libraries

ACM Reference Format:
Hailie Mitchell. 2022. Automatically Fixing Breaking Changes of Data Science Libraries. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3551349.3559507

1 INTRODUCTION AND BACKGROUND

There has been significant growth in the use of data science libraries. For machine learning libraries specifically, usage in new projects on GitHub increased from less than 2% in 2013 to nearly 50% in 2018 [1]. These libraries have frequent new releases that often include breaking changes or bug fixes, so developers must continuously update projects accordingly [6]. However, updating source code to newer versions of libraries is difficult, time-consuming, and error-prone, deterring developers from making updates [1].

One commonly reported challenge is updating source code to adapt to breaking changes and deprecation, and many developers seek code update tools to automate the process [1]. Previous work inferred Comby templates to apply type changes [3]. However, this was limited to only Java type changes and required a large training dataset of 400K commits to infer rewrite rules. Other work, including APIMigrator [2], A3 [4], and Meditor [9], also depend on updated example code from other developers for pattern inference and are not applicable to non-object-oriented languages like Python. Unlike previous work, our tool infers rules to update Python programs using old APIs. Furthermore, CombyInferPy mines pull requests from GitHub repositories of libraries that introduce a breaking change or deprecation to infer a set of Comby rules describing the transformation. Inferring rules directly from library source code minimizes the likelihood of incorrect API usage and allows users to use the tool without relying on other developers' updates or collecting large training data. Figure 1 shows an example of a code change from a GitHub pull request that deprecates the "`hide_index`" method in pandas version 1.3.5 and replaces it with a new "`hide(axis=..)`" method for version 1.4.0. CombyInferPy uses the code change to infer a Comby rule, shown in Figure 2 [8].



```
-     ctx = mi_styler.hide_index(level=level)._translate(False, True)
+     ctx = mi_styler.hide(axis="index", level=level)._translate(False, True)
```

Figure 1: GitHub commit of pandas API update from hide_index (Code A) to hide(axis=..) method (Code B) [5].

```
:[[a]] = :[[d]].hide_index(level=:[[e]]).:[[b]](:[f], :[g])
:[[a]] = :[[d]].hide(axis='index', level=:[[e]]).:[[b]](:[f], :[g])
```

Figure 2: Comby rewrite rule inferred from pandas update.

Once a set of rewrite rules is inferred from a given pull request, Comby applies them to a specified file or files [8]. Code that structurally matches the old API in a Comby rule is transformed to the new API. We evaluate our approach on four pandas pull requests from two library releases, and preliminary results show the applied rules resolve several deprecation warnings.

2 METHODOLOGY

An overview of the CombyInferPy pipeline is shown in Figure 3. The input for the tool is two code snippets from which rewrite rules are derived. To generate code snippets, we take as input a data science library pull request number, an old API call, and, optionally, a replacement call. Changed lines in the pull request are automatically isolated, then saved to a new directory if they contain the specified deprecated and replacement call. For example, in Figure 1, our program searches for a call to the method "`hide_index`" in the old code and for a call to the method "`hide`" in the corresponding new code. Since the code snippets in Figure 1 match these conditions, they are automatically saved to later be used for rule inference.

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559507

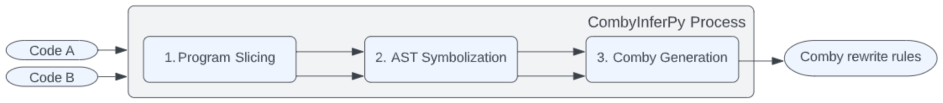


Figure 3: Overview of CombyInferPy process.

Once all of the relevant changed line pairs have been saved to a new directory, CombyInferPy infers a rewrite rule for each pair of code snippets. This inference process occurs in three stages.

Program Slicing: The CombyInferPy tool works by first pre-processing the code example with program slicing. This eliminates any identical lines between the before and after code snippets so only the relevant changes are incorporated into a rule.

AST Symbolization: The processed code snippets are parsed into abstract syntax trees, or ASTs. Several of the AST nodes have an identifier attribute derived from variable or method names. The AST associated with the before code snippet from Figure 1 is shown on the left side of Figure 4. The tool individually traverses each of the ASTs and replaces node identifiers with template variables, symbolized by letters, illustrated on the right side of Figure 4.

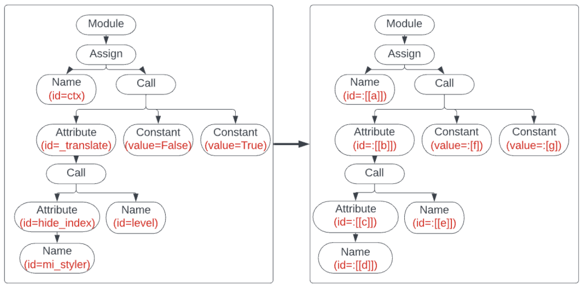


Figure 4: AST Symbolization sub-process for Code A.

Comby Generation: Comby is a tool to search and transform code of any programming language. These syntactic transformations are accomplished with lightweight templates that match general code structure [8]. To use Comby, a rule must have a match template and a rewrite template. The match template is the code structure for which Comby searches. The rewrite template describes how to restructure the matched code based on the variables in the match template [7]. In Figures 2 and 5, the top line of the Comby rule is the match template, derived from code A, and the bottom line is the rewrite template, derived from code B. In Comby syntax, a template variable is represented by a variable name surrounded by a set of square brackets and preceded by a colon.

```
:[[a]] = :[[d]].:[[c]](level=:[[e]]).:[[b]](:[f], :[g])
:[[a]] = :[[d]].:[[h]](axis=:[i], level=:[[e]]).:[[b]](:[f], :[g])
```

Figure 5: Initial templates in Comby generation.

Once CombyInferPy replaces AST node identifiers with template variables, the tree is unparsed into an intermediate Comby template. Any part of the code that is not a Python keyword or symbol is replaced with a Comby template variable, shown in Figure 5. CombyInferPy then examines maps relating template variables to their replaced identifiers. By comparing the maps of code A and B snippets, template variables that are not common between both snippets are changed back to the concrete value of the original identifiers. Template variables that are common between code snippets remain as variables. The result of this step is seen in Figure 2.

Once a rule is inferred, CombyInferPy applies the rewrite rule to code A and ensures it exactly matches code B. If the resulting code matches as expected, the rewrite rule is compared to other rules from the same pull request to prevent duplicates. The output of the tool is a set of similar rewrite rules in the form of Comby templates that update the same API in different instances. The set can then be applied to projects that use older versions of the library to automatically update them for newer versions.

3 RESULTS AND DISCUSSION

We evaluated the CombyInferPy tool by running the pandas version 1.3.5 tests on the version 1.4.0 library and found 3606 failed tests and 3120 warnings. We then applied sets of rewrite rules for three API updates on the version 1.3.5 tests and ran the updated tests with the 1.4.0 library code to compare the number of failed tests to before the rule was applied. The same was done for two API updates in the version 1.3.0 library update. Note that these failing tests and warnings are related to all APIs from the pandas library. Since this preliminary evaluation only focuses on updating five APIs, we do not expect to fix a large number of tests or warnings since only a fraction of the total failures are related to these APIs. Table 1 shows the number of rules applied, warnings resolved, and tests fixed for each updated API call.

Table 1: Results for pandas updates

PR #	Updated API	# Rules	Warnings	Tests
42140	render –> to_html	22	61	0
45076	pad –> ffill	2	7	0
43771	hide_index –> hide	11	13	0
38701	lexsort_depth –> _lexsort_depth	5	8	0
38701	is_lexsorted –> _is_lexsorted	4	34	0

Challenges and Limitations: Our work focuses on breaking changes between consecutive library releases involving a single line of code, but we expect the process can be generalized to more complex changes. We have also observed that in the set of rules inferred by CombyInferPy, some rewrite rules are not identical, but have the same effect. For example, the rule in Figure 2 can generalize to `.hide_index(:[a]) --> .hide(axis='index', :[a])` by eliminating identical information and further abstracting arguments. So, there is an opportunity to automatically generalize the set of rules to fewer, more general rules that might match more diverse API usage. To increase the tool's precision, we also plan to incorporate type information in rule inference and application.

One limitation of our current evaluation is that it only analyzes failing tests. For stronger results, we must check that passing tests are not affected by the Comby updates. Moving forward, we also plan to infer rules for other Python libraries beyond pandas.

REFERENCES

[1] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 55 (July 2021), 42 pages. https://doi.org/10.1145/3453478

[2] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2020. APIMigrator: An API-Usage Migration Tool for Android Apps. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems* (Seoul, Republic of Korea) *(MOBILESoft '20)*. Association for Computing Machinery, New York, NY, USA, 77–80. https://doi.org/10.1145/3387905.3388608

[3] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryskin. 2022. Inferring and Applying Type Changes. In *44th International Conference on Software Engineering* (Pittsburg, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1206–1218. https://doi.org/10.1145/3510003.3510115

[4] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering* 48, 2 (Feb. 2022), 417–431. https://doi.org/10.1109/tse.2020.2988396

[5] pandas development team. 2022. GitHub commit of pandas API update from $hide\_index$ and $hide\_columns$ to $hide(axis = ..)$. https://github.com/pandas-dev/pandas/pull/43771. Accessed: 08-16-2022.

[6] Jeff H. Perkins. 2005. Automatically generating refactorings to support API evolution. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal) *(PASTE '05)*. Association for Computing Machinery, New York, NY, USA, 111–114. https://doi.org/10.1145/1108792.1108818

[7] Rijnard van Tonder. 2022. *Comby*. Retrieved July 19, 2022 from https://comby.dev/docs/overview

[8] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI '19)*. Association for Computing Machinery, New York, NY, USA, 363–378. https://doi.org/10.1145/3314221.3314589

[9] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension* (Montreal, QC, Canada) *(ICPC '19)*. IEEE, 335–346. https://doi.org/doi:10.1109/ICPC.2019.00052