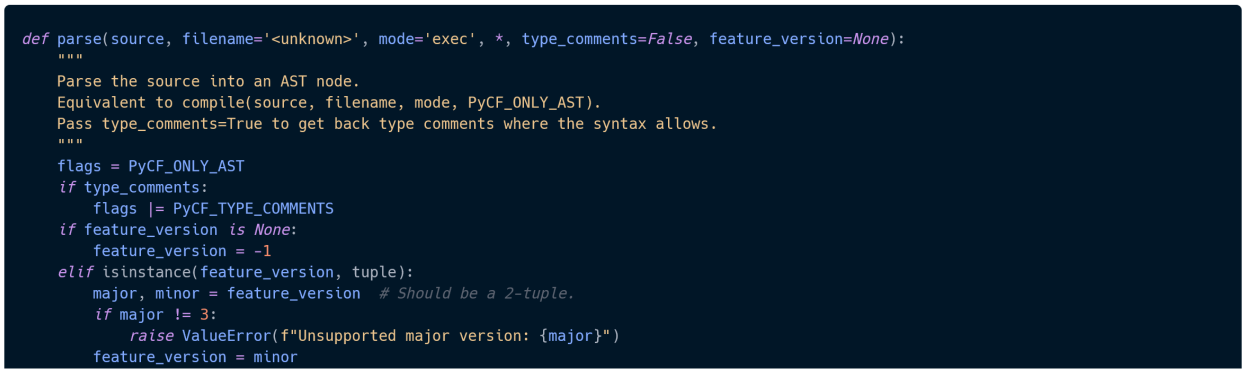
Calculating the Modernity of Popular Python Projects

CHRIS ADMIRAAL, University of Twente, The Netherlands



```python
def parse(source, filename='<unknown>', mode='exec', *, type_comments=False, feature_version=None):
    """
    Parse the source into an AST node.
    Equivalent to compile(source, filename, mode, PyCF_ONLY_AST).
    Pass type_comments=True to get back type comments where the syntax allows.
    """
    flags = PyCF_ONLY_AST
    if type_comments:
        flags |= PyCF_TYPE_COMMENTS
    if feature_version is None:
        feature_version = -1
    elif isinstance(feature_version, tuple):
        major, minor = feature_version  # Should be a 2-tuple.
        if major != 3:
            raise ValueError(f"Unsupported major version: {major}")
        feature_version = minor
```

Fig. 1. Official Python code for parsing code into an AST

Python has undergone a lot of (syntax) changes throughout all its versions. We present a static analysis method which calculates a *modernity signature* for a Python project, which determines its age. This also allows us to discuss how often and quick the adoption of a certain Python version/feature is by developers.

Additional Key Words and Phrases: Python, Modernity, Vermin, AST

1 INTRODUCTION

The programming language Python has undergone a lot of changes throughout the past 20+ years. The most (breaking) changes occurred when major version 3 came out [15]. For example: `print` was not a keyword anymore, but a function; behaviour in integer division changed; and strings were Unicode by default instead of ASCII. In contrary to minor versions, major versions of Python are not backward compatible.

Recently in Python 3.10 new syntax features have been added. The most important being, Structural Pattern Matching [9]. All these new (syntax) features allow programmers to write their code structurally differently.

With all these (syntax) changes to the language, there are multiple ways of writing code that functionally behave the same. So, writing the same algorithm 10 years ago will (most likely) look different then if it was written today with the newest features. This research focuses on this difference, any may tell us how old, and maybe therefore how maintained, a particular Python project is.

Similar research has been in the PHP domain [19], which we would like to replicate in Python. In that research one grammar was constructed (usable for all PHP versions) to generate an AST (Abstract Syntax Tree) which was used to generate the modernity signature.

The first problem arises from the fact that we would like to compare Python 2 and Python 3 code in this research. This way we can see how many features that were introduced in Python 2, are still used nowadays (compared to the past). To compare both versions, we would like to have one grammar that parses both languages. Grammars that parse Python 2 and Python 3.0 till 3.9 do already exists. For example, Vavrová and Zaytsev used a combination of two existing ANTLR grammars, but the Python 3 grammar has not been updated at this point in time [4, 22]. Also, third party grammars such as *Parso*, which parses both Python 2 and 3 code into the same AST, have not succeeded to do so [17]. This AST differs from the built-in `ast` Python module [16].

The main reason these grammars do not support Python versions 3.9 and above, is the introduction of the PEG (Parsing Expression Grammar) parser in Python 3.9 which replaced the LL(1)-based parser. A grammar can be said to be LL(1) if it can be parsed by an LL(1) parser, which in turn is defined as a top-down parser that parses the input from left to right, performing leftmost derivation of the sentence, with just one token of lookahead [21]. A PEG grammar differs from a context-free grammar (like the old Python grammar) in the fact that the way it is written more closely reflects how the parser will operate when parsing it. The fundamental technical difference is that the choice operator is ordered. And in Python 3.10 new keywords *match* and *case* were introduced, which were made soft, so that they are recognized as keywords at the beginning of a match statement or case block respectively but are allowed to be used in other places as variable or argument names [2]. But this means the current Python grammar may not (trivially) be written in a LL(1) grammar.

Presumably this PEG-problem could be solved by convergence of both major grammars [11, 23], which has been done before with Python [12]. But to the best of our knowledge, such research has

TScIT 38, February 3, 2023, Enschede, The Netherlands

not been done yet with the latest Python 3 grammar. As this might not be a trivial task and solving this problem is not the focus of this research, we will opt for a different approach to calculate the modernity signature.

1.1 Research Question

From the above-mentioned problems, the following research question can be derived:

To what extent can we use static analysis methods to reliably detect the modernity of a Python project?

With the following sub question:

RQ1 How can we define a modernity signature of a Python project?

2 RELATED WORK

How modern a Python project is, may also be related to the occurrence of pythonic idioms. Many of such idioms are also features, introduced in a specific Python version. Research already has been done on which and how much a particular idiom is popular in a certain time period [5]. The idiom-detector in this research makes use of the built-in `ast` module.

In many other works that do some static analysis on the Python language, we see the use of this module to do analysis on the constructed AST. For example: Peng et al. used it to investigate certain language features in Python projects from different application domains [14] and Chen et al. traversed it for smell detection [3]. Such research also exists outside the Python domain, for example in Java [13].

3 METHODOLOGY

Just like van den Brink et al., we define modernity as a scale of measuring the age of a project. The modernity signature takes the form of an n-tuple with n being the number of minor Python 2 & 3 versions. Currently there are 20 such versions available (2.0 - 2.7 and 3.0 - 3.11). Every element in this tuple is going to represent the number of features that are used in the given project, which are introduced by this Python version. To compare different releases of the same project, we normalize this tuple by dividing all its elements by the total sum of this tuple.

To calculate this modernity signature, we will build a tool called `Pyternity`. To generate the modernity signature for any given project, we need to know how many features per Python version there are in a project. To the best of our knowledge, there currently only exists one up-to-date tool that achieves this: *Vermin* [10]. This tool claims to be able detect the minimum Python version (major and minor) needed to run any given Python code. It is also able to tell us why it requires a certain version: it will return a list of features that *violate* the given version that are used in the given code sample. It achieves this by traversing the built AST and detecting if predefined rules for features match.

So, to know how many features per Python version there are in a project, we first download the project from PyPI and only extract all Python files. Secondly, we (concurrently) iterate over all these files and call Vermin to return all features (with the Python version it was introduced in) present in these files. However, some features are introduced in two different Python versions (for example `collections.Counter` was introduced in both Python 2.7 and 3.1), we count both. The results of all these files are first mapped to the Python version it was introduced in, then mapped to the feature's name with the number of times it occurs in the whole project. This mapping is also saved to file. By summing up all (non-unique) features per Python version and dividing it by the total number of features detected in this project, we now have generated the modernity signature.

4 EXPERIMENT

The exact Python implementation, all raw data and graphs can be found on https://github.com/cpAdm/Pyternity. In the `README.md` file one can find more information on which commands to run, to reproduce the experiment.

4.1 Environment

All experiments and tests listed in this research are run with Python 3.11.0 on a Windows 10 (21H2) machine with 16GB of RAM and an 8-threaded Intel® Core™ i7-7700HQ processor. However, when running the program on projects introduced later then Python 3.11.0, one should use the latest Python and Vermin version to detect new features.

4.2 Data

The experiment is run on the 50 most downloaded PyPI projects (as of 01-01-2023), gathered from *Top PyPI Packages* [20]. To save resources, only for all minor versions of these projects the signature is calculated. All this data is retrieved from PyPI's JSON API [6].

Not all (versions of) these projects are applicable for this research. Projects that do not contain Python files are not considered. This can for example be the case if a project is (mainly) written in C, or when only its build distribution is available and not its source distribution.

4.3 Results

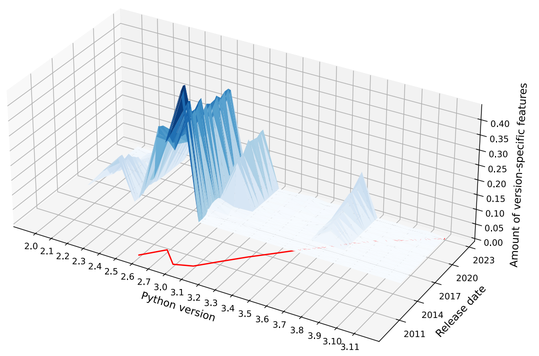
We run `Pyternity` on the aforementioned environment and data, which takes about 2 hours to process all 146.362 Python files (1.585 MiB) in the 1.570 releases. We can detect a lot of different errors. To begin with, there are a lot of edge cases the tool does not handle correctly. For example, there are 1.051 Python files where Vermin is in conflict; it is not able to tell which minimum Python version is required. It detects Python2-only features and Python3-only features in the same file. But this is off course not possible (assuming the given source code is valid).

Secondly, some Python files threw errors during Vermin's feature detection. A `RecursionError` occurred for versions 0.2 - 2.0 of *idna*. And a `TypeError` occurred for *pandas* 1.5.0.

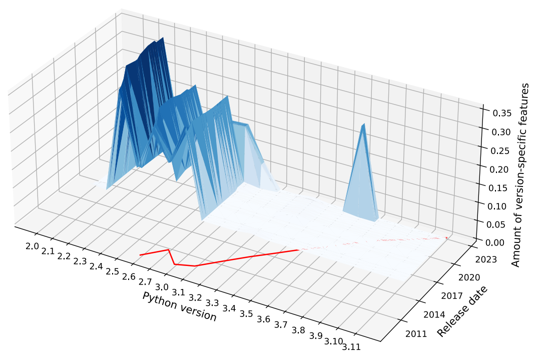
4.4 Discussion

All the signatures of the 50 projects have been combined into one graph, see Figure 3. Figure 2 shows six projects that have been handpicked to discuss generic trends and exceptions in these trends.

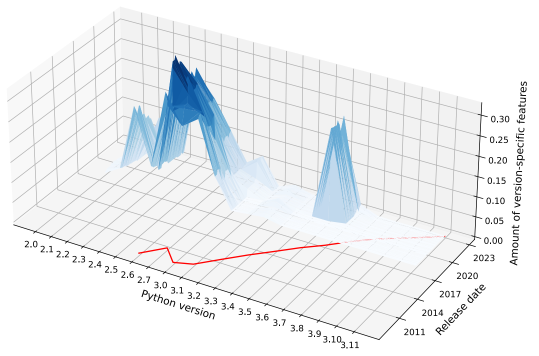
In all these graphs a red line has been plotted, which shows the minor Python releases through time. Do note the drop after 2.7,



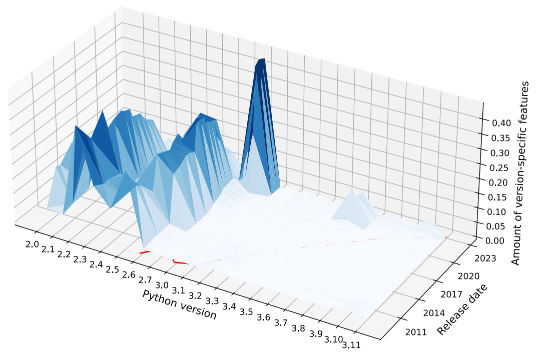
(a) Attrs



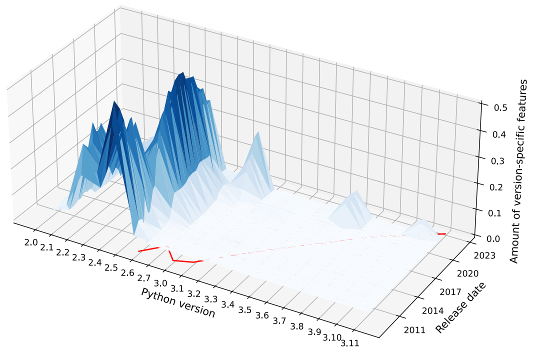
(b) Boto3



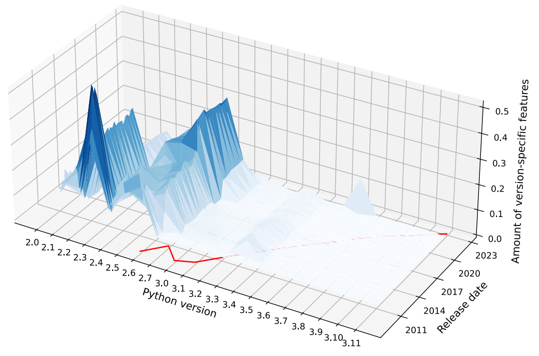
(c) google-api-core



(d) NumPy



(e) pandas



(f) Requests

Fig. 2. Modernity signatures of different popular Python projects

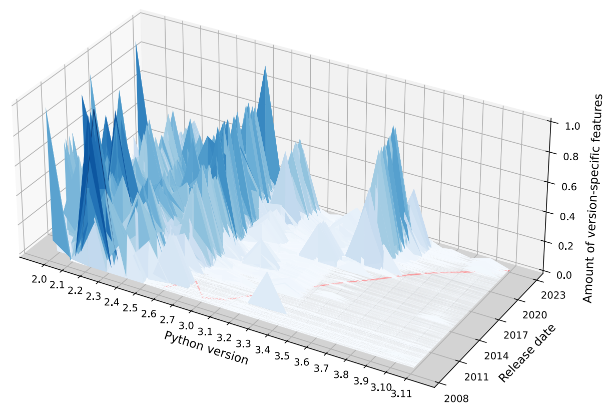


Fig. 3. All modernity signatures of the 50 projects combined

since Python 3.0 and 3.1 released before Python 2.7. It should not be possible for any data to be shown to right of this line, as one cannot use features of a Python version that has not been released yet in that point in time. That is, the release date of the minimum Python version required to run this project's release, should not exceed the release date of this project's release. However, we do see some data for Python 3.2 - 3.4 before its release. The incorrect peak at 3.3 is caused by *urllib3* and *requests* (Figure 2f). Vermin detects a `TimeoutError` (Python 3.3 feature). But when looking in the source code of these projects, it turns out the developers defined a `TimeoutError` class themselves. This is also the case for a couple of other wrongly identified features. This is a limitation of (almost) only using the AST for analysis.

In general we see that a lot of Python 2 features are still used nowadays, although its trend differs wildly per minor version and per project. After 2017, we do see an increasing use of Python 3.5 features. Besides the introduction of the `typing` module and *Additional Unpacking Generalizations*, the most common feature detected for this Python version are the coroutines `async` and `await`. This is for example clearly visible for *google-api-core* in Figure 2c. This observation is also supported by its changelog which mentions this AsyncIO integration [8].

The peak at 3.6 in Figure 2b for *Boto3* it is the sudden introduction of *f-strings* to the codebase. This is also the case for *attrs* (Figure 2a), with the addition of also using *variable annotations*, even quite quickly after Python 3.5 was released.

Another interesting finding is that some projects decide to use features in their code that have not been released. This is for example the case for *typing_extensions* and *setuptools*. However, when looking at the code, we do find an if-clause surrounding the used feature, see Listing 1.

Out of the 2.938.792 detected features, the most common feature is the `with` statement (Python 2.5), detected 528.769 times in total.

```
if hasattr(typing, 'Required'):
    Required = typing.Required
    NotRequired = typing.NotRequired
elif sys.version_info[:2] >= (3, 9):
    ...
```

Listing 1. Part of source code of src/typing_extensions.py in typing_extensions 4.4.0

Second is the *function decorator* (Python 2.4, 353.749 times) and third *byte strings* (Python 2.6, 224.301 times).

5 THREATS TO VALIDITY

The biggest threat in this research is the use of Vermin to calculate the modernity signatures. To determine how accurate this tool is, we will test which and how many features it is able to correctly detect. In the first subsection we describe how we achieve this content validity. In the second subsection we describe how we ensure population validity such that we can generalize our findings.

5.1 Validation of Vermin

To validate if Vermin actually detects all new features for each Python release, we should write a test case for each such a new feature. To obtain all these new features, one could look through all Python's changelogs. But not everything that has been added in a new Python version, is (explicitly) listed here. For example, the function `io.text_encoding` was added in Python 3.10, but only its PEP 597 was listed in the changelog. However, the addition of this new function is listed on the documentation page of the `io` module with *"New in version 3.10."*. Similarly, the addition of new parameters is denoted with *"Changed in version X.Y."*. So, to obtain new features, we will iterate through all the library's documentation and look for this text. Since this is a very time-consuming task (Python 3 documentation has thousand occurrences of such text), we automate this process.

To find these new features, we first parse Python's source documentation using *Sphinx* (Python's documentation generator) into machine readable *doctrees*. Each doctree represents a documentation page. Then by adding our own Sphinx extension, we traverse the built doctree and look for all *versionadded* and *versionchanged* nodes in this tree. For each of these nodes, we then may generate a test case. So, test cases are generated if a new module, function, class, method, parameter, constant, attribute or exception has been added.

Doing this for the latest Python 2 (2.7.18) and Python 3 (3.11.0) documentation, we generate 2.403 test cases within 30 seconds after the doctrees have been built. The results of this test can be seen in Figure 4 and Table 1. After manually removing all invalid test cases, 169 tests fail. We have reported these test failures to Vermin [1].

In terms of percentage the most fail for the oldest and newest Python versions. There are less test cases generated then the 3.556 features Vermin can detect. Due to several reasons test cases are missing or invalid. The biggest oversight by generating test cases this way, is that there is no test coverage for Python 3.0 (see drop/spike at Python 3.0 in Figure 4) because the Python 3 documentation does not mention major version changes in its library documentation.

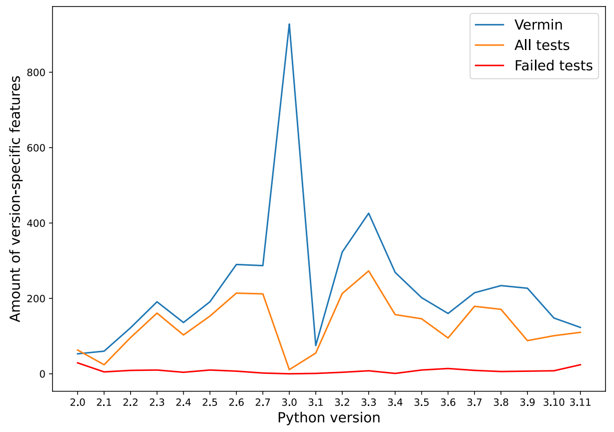


Fig. 4. Detected features by Vermin versus verification

Also, it is not trivial to determine if a specific *versionchanged* node is describing a new parameter or for example a behavioural change, since there is no consistency in its explanation. This is also not enforced by the devguide of Python [7]. Furthermore, the Python documentation itself contains mistakes. For example, in the `_wingreg` module (Python 2.7.18), *"New in version 2.7."* is wrongly indented for the functions `CreateKeyEx` and `DeleteKeyEx`. So, there will be no test cases generated for these functions. The Python documentation also does not cover all its code base, this overview can be generated by building the Python documentation with target `coverage`.

The automatically generated test cases are limited to library features and do not generate test cases for syntax features. Vermin does detect syntax features like 78 built-in typing annotations. This also explains the especially low test coverage for Python 3.9 as can been seen in Figure 4, since these features are introduced in this Python version. To see which (other) syntax features Vermin detects, some manual test cases have been written (by going through "What's New in Python" for all versions). Vermin indeed only detects the syntax features listed in its `README.rst`. For example, type hinting generics in standard collections (PEP 585) are not detected.

5.2 Population validity

The experiment is run on the 50 most popular PyPI projects and only 6 projects are discussed. We make our results more representative by also picking projects that are started later in time but are still of significance for the Python community. As stated before, the full data set and all graphs can be found on the aforementioned GitHub repository. In addition, `Pyternity` has an extensive command line interface such that one also run the tool on any given (popular) PyPI project.

6 CONCLUSION AND FUTURE WORK

We have built the tool `Pyternity` for calculating the modernity signature for a Python project. It uses Vermin under the hood to detect version specific features. We use this information to construct the n-tuple signature, normalized by the total number of features detected in the project. Some trends have been detected by calculating this signature for popular Python projects. More research could be done to dive deeper into these trends to see which specific features cause this trend, data generated by this research could be used for that.

Furthermore, there is room for improvement for Vermin and the verification of it. We have already given cases in which it was not able to detect certain syntax/library features. A novel idea would be to look at the Python source code to generate these verification test cases instead of the error prone documentation. However, some Python modules are written in C, making it less trivial to generate test cases. But stub files in contrary, used for type hinting, are also present for such modules and even contain version specific information. These files for the Python standard library can be found in the official Python *typeshed* repository [18].

REFERENCES

[1] Chris Admiraal. 2023. *Library features that are not (correctly) detected · Issue #144 · netromdk/vermin.* https://github.com/netromdk/vermin/issues/144

[2] Brandt Bucher, Daniel F Moisset, Tobias Kohn, Ivan Levkivskyi, Guido van Rossum, and Talin. 2020. *PEP 622 – Structural Pattern Matching.* Python Software Foundation. https://peps.python.org/pep-0622/

[3] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE).* IEEE, Kunming, China, 18–23. https://doi.org/10.1109/SATE.2016.10

[4] dours. 2021. *Do you have plans to support python 3.10 (for example, assignment expressions)? · Issue #2462 · antlr/grammars-v4.* https://github.com/antlr/grammars-v4/issues/2462

[5] Aamir Farooq and Vadim Zaytsev. 2021. There is More than One Way to Zen Your Python. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering* (Chicago, IL, USA) *(SLE 2021).* Association for Computing Machinery, New York, NY, USA, 68–82. https://doi.org/10.1145/3486608.3486909

[6] Python Software Foundation. 2023. *PyPI JSON API.* https://warehouse.pypa.io/api-reference/json.html

[7] Python Software Foundation. 2023. *Python Developer's Guide: reStructuredText Markup.* https://devguide.python.org/documentation/markup/#paragraph-level-markup

[8] Google. 2023. *Changelog — google-api-core documentation.* https://googleapis.dev/python/google-api-core/latest/changelog.html#id118

[9] Tobias Kohn and Guido van Rossum. 2020. *PEP 635 – Structural Pattern Matching: Motivation and Rationale.* Python Software Foundation. https://peps.python.org/pep-0635

[10] Morten Kristensen. 2018. *Vermin.* https://pypi.org/project/vermin/

[11] Ralf Lämmel and Vadim Zaytsev. 2009. An Introduction to Grammar Convergence. In *Integrated Formal Methods*, Michael Leuschel and Heike Wehrheim (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 246–260.

[12] Brian A. Malloy and James F. Power. 2017. Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, Toronto, ON, Canada, 314–323. https://doi.org/10.1109/ESEM.2017.45

[13] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2011. Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (Waikiki, Honolulu, HI, USA) *(MSR '11).* Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1985441.1985446

[14] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, Honolulu, HI, USA, 24–35. https://doi.org/10.1109/SANER50967.2021.00012

[15] Python. 2008. *What's New In Python 3.0.* Python Software Foundation. https://www.python.org/download/releases/3.0/whatsnew

[16] Python. 2022. *ast — Abstract Syntax Trees.* Python Software Foundation. https://docs.python.org/3/library/ast.html

[17] Batuhan Taskaya. 2020. *Soft Keywords and How to Implement Them · Issue #138 · davidhalter/parso.* https://github.com/davidhalter/parso/issues/138

[18] The Python Typing Team. 2023. *typeshed.* https://github.com/python/typeshed/tree/main/stdlib

Table 1. Verification failures per version

Python version	**2.0**	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**	**2.7**	**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**3.9**	**3.10**	**3.11**
% of tests failed	46.03	20.83	9.38	6.21	3.88	6.54	3.27	0.94	0.00	1.82	1.88	2.93	0.64	6.85	14.74	5.03	3.51	7.95	7.92	21.82

[19] Wouter van den Brink, Marcus Gerhold, and Vadim Zaytsev. 2022. Deriving Modernity Signatures for PHP Systems with Static Analysis. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Limassol, Cyprus, 181–185. https://doi.org/10.1109/SCAM55253.2022.00027

[20] Hugo van Kemenade, Richard Si, and Zsolt Dollenstein. 2023. *hugovk/top-pypi-packages: Release 2023.01*. https://doi.org/10.5281/zenodo.7497599

[21] Guido van Rossum, Pablo Galindo, and Lysandros Nikolaou. 2020. *PEP 617 – New PEG parser for CPython*. Python Software Foundation. https://peps.python.org/pep-0617

[22] Nicole Vavrová and Vadim Zaytsev. 2017. Does Python Smell Like Java? *The Art, Science and Engineering of Programming (‹Programming›)* 1 (April 2017), 11–1–11–29. Issue 2. https://doi.org/10.22152/programming-journal.org/2017/1/11

[23] Vadim Zaytsev. 2013. Guided Grammar Convergence. In *Poster proceedings of the Sixth International Conference on Software Language Engineering (SLE 2013)*. Springer International Publishing, Indianapolis, IN, USA, 117–136.