There Is More Than One Way to Zen Your Python

Aamir Farooq
University of Twente
Enschede, The Netherlands
aamir@grammarware.net

Vadim Zaytsev
University of Twente
Enschede, The Netherlands
vadim@grammarware.net

Abstract

The popularity of Python can be at least partially attributed to the concept of *pythonicity*, loosely defined as a combination of good practices accepted within the community. Despite the popularity of both Python itself and the pythonicity of code written in it, this concept has not been studied that well, and the first attempts to define it formally are rather recent. In this paper, we take the next steps in exploring this topic by conducting an independent literature review in order to create a catalogue of *pythonic idioms*, reproduce the results of a recent paper on the usage of pythonic idioms, perform an external direct replication of it by reusing the same open source toolset and dataset, and extend the body of knowledge by also analysing how the use of pythonic idioms evolve over time in open source codebases.

CCS Concepts: • **Software and its engineering → Patterns**; Scripting languages.

Keywords: Python, pythonicity, pythonic idioms

ACM Reference Format:
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 (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3486608.3486909

1 Introduction

A software language is not only its syntax and semantics, but also a set of known effective ways to solve actual problems with it. Some authors refer to this elusive part of a software language as its *pragmatics* [8, 79], but it is an overloaded term very frequently used to refer to language implementation [59, 74]. Studying the *syntax* of a successful language could help to find better ways to express language concepts. Studying the *semantics* potentially leads to discovering new concepts or alternatives to existing ones. Studying language *implementation* is also useful since it helps to make languages more efficient and to reach the Holy Grail of domain-specific languages [73] and language-oriented programming: rapid language prototyping. However, studying those "ways to solve actual problems" may lead us to the real cause of popularity and success of some languages and the esoteric and obscure reputation of others. For the lack of an even better term, we will use *traditions* to refer to idioms, conventions, styles and patterns.

Software language traditions have been studied fairly well — we present some insights into the existing body of knowledge on them in the next section. It is worth noting that many claims about coding traditions that seem reasonable, are much more complex upon closer inspection. For instance, coding styles associated with expert developers, are not always perceived as more readable and understandable by language learners, and learning to write in an idiomatic style does not guarantee the development of a preference to find idiomatic code more readable [77]. Adherence to a certain convention also does not necessarily grow as the project matures: quite often it remains stable, varies per convention and for some even declines over time [40], and even systematic code review does not stop the growth of violations [27], the number of which simply yet inevitably grows together with the LOC count [67].

The prevalence of coding traditions in software engineering practice being indisputable, we still proceed with caution. We zoom in on Python, one of the most popular programming languages, where the concept of pythonic idioms as "reusable abstractions" has been defined and studied by Alexandru et al. [1]. Python developers call code *pythonic* when such idioms are used. The *pythonicity* of a piece of code stipulates how concise, easily readable, and in general terms, "good" the code is. This concept of pythonicity and the concern of code being pythonic or not pythonic, is especially pronounced in the Python community. There is a general "feeling" shared among the community that it goes beyond a set of practices — it is rather a philosophy that the community strives to uphold. Python developers are in constant pursuit of upholding the so-called *Zen of Python* rules, such as "*There should be one — and preferably only one — obvious way to do it.*", and "*Beautiful is better than ugly. [...] Simple is better than complex.*" [43]. Not all these rules are immediately actionable, implementable and detectable.

Given a piece of code, any experienced Python developer can easily tell whether it is pythonic or not. Sakulniwat et al. were able to demonstrate, in a case study of the `with open` idiom, that in Python developers tend to adopt idioms

SLE '21, October 17–18, 2021, Chicago, IL, USA

over time to improve their codebase [55] (as we have stated above, this observation does not generalise well over languages and even over conventions within one language). In the interviews conducted by Alexandru et al., experienced developers stated that year after year, their code became more pythonic [1]. However, to programming novices or newcomers to Python, as Alexandru et al. also contend, it is not completely obvious how to incorporate the so-called *pythonic idioms* in their code [1]. In their study, many interviewees also indicated that junior Python programmers can even be distinguished from more experienced ones simply by observing the usage of pythonic idioms, and further, the interviewees agreed that they learnt pythonicity from experience — by reading books, source code from other projects and StackOverflow responses [1].

Since Alexandru et al. identified a lack of research in the phenomenon of pythonicity as they felt that there was no clear definition as to what "pythonic" means and what should developers do to make their code pythonic. They conducted a literature review to identify the pythonic idioms from numerous sources such as *The Zen of Python* [43], *Writing Idiomatic Python* [31], *The Hitchhiker's Guide to Python* [51], *Effective Python* [65], *The Little Book of Python Anti-Patterns* [49], as well as direct interviews with developers with varying levels of expertise. They wrote an idiom detection library to support the claim that idioms were in use in the most popular open-source Python projects on GitHub. The library is available at https://github.com/jjmerchante/Pythonic, and we report below on our activities to reproduce the results.

To guide our investigation, we have devised the following research questions which are intended to be answered and commented on by the end of this paper. Our null hypothesis is that the popularity of each idiom did not change between 2018 and 2021. (It is based on the sentiment from the developers Alexandru et al. interviewed, that they do *not* go back and make their old code pythonic [1]).

RQ1: *What idioms should be included in an updated, extended catalogue of pythonic idioms?*
By reinferring and updating the catalogue of idioms that Alexandru et al. have identified based on their literature review from Python books, we can form a comprehensive picture of what idioms make code pythonic.

RQ2: *How widely adopted are the new idioms?*
We will also need to find empirical evidence to support the claim that both reconfirmed and newly documented idioms are accepted as pythonic by the Python community. This means extending the idiom detection code of Alexandru et al. to include the newly found idioms and analysing the statistics we find.

RQ3: *How has the usage of pythonic idioms evolved in software projects over time?*
As stated previously, some years have passed since the experiment of Alexandru et al. It is possible that among the idioms they have identified and counted, certain idioms have gone out of style and other, possibly new, idioms, have become more popular. By answering this question, we can provide empirical evidence to not only support the results of **RQ2** but also to comment on our hypothesis.

The rest of the paper is structured as follows. In § 2 we proceed with the preliminaries and go deeper into the field of coding traditions, their prevalence across software languages, and impact on the practices and the practitioners of software engineering. In § 3, we answer **RQ1** by performing a systematic review of existing literature about pythonicity, pythonic idioms and recipes. As a result, we compose a catalogue of 46 pythonic idioms, combining 21 reconfirmed ones with 25 newly identified traditions. In § 4, we perform a full external replication of the idiom detection work by Alexandru et al. [1], since that not only strengthens and verifies the original claims, but gets us closer to answering **RQ2** and **RQ3**. In § 5, we go beyond the straightforward replication and perform detection of a subset of pythonic idioms on a new set of code repositories. Again, this helps to shed light on **RQ2** and provides us with tools and data to tackle **RQ3**. Then, we perform a new experiment with the freshly collected data, in order to analyse how the usage of pythonic idioms evolves over time in different repositories, and report on a small selection of illustrative idiom evolution patterns. § 6 concludes the paper by summarising its findings and explicitly answering the research questions defined above.

2 Related Work

The oldest paper on idiomatic code patterns is dated 1987: it concerned APL [41] and considered "collections of symbols commonly recognised as a useful building block" which were analysed by "scales" (metrics). Since that time, there have been around 500 papers published on coding conventions, calling conventions, naming conventions, formatting conventions, coding styles, coding guidelines, idioms, implementation patterns, antipatterns, nanopatterns, micropatterns, code snippets and idioms. The full analysis of them is beyond the scope of the paper, but we would like to note that these papers covered idioms in almost all existing software languages from assembly [54] to C++ [11], from Erlang [58] to Haskell [36], from UML [13] to EBNF [81], from C# [15] to F# [57], from SQL [39] to OWL [53], from Javascript [9] to CSS [26], from COBOL [14] to PL/I [6]. They have been investigated for the obvious reasons such as readability and other aspects of maintainability, but also in the context of security [28], performance [61], code review [27], teaching novices [78], software migration [33], etc. Attempts were made to solve the problem with DSLs like CCL [4], a calling convention specification language. One of the foci of modern research on coding traditions is to use code snippets as somewhat parametric pieces of idiomatic code for solving

```python
animal_one = "fox"
animal_two = "dog"

# non-pythonic!
print("The quick brown " + animal_one + " jumped over the lazy sleeping " + animal_two)

# the "ancient" pythonic!
print("The quick brown %s jumped over the lazy sleeping %s" % (animal_one, animal_two))

# the "old" pythonic!
print("{foo} is {bar}.".format(foo="Resistence", bar="futile"))
print("The quick brown {0} jumped over the lazy sleeping {1}".format(animal_one, animal_two))

# the "new" pythonic!
print(f"The quick brown {animal_one} jumped over the lazy sleeping {animal_two}")
```

Figure 1. An example of a new pythonic idiom Alexandru et al. did not cover, known as f-strings, a much less cumbersome and more readable approach to traditional string formatting methods [69]. The code above shows the language features slowly improving and becoming more powerful and expressive, but also adapting themselves to the programming practices. The first variant is based on string concatenation and required either a `str()` call for non-string values or backticks that implicitly called `repr()`. It is cumbersome to use for the developer, as well as inefficient for the interpreter. The second variant is positional formatting, which used the percent symbol, inspired by the classic `%s` and similar formatters for `printf` and `scanf` in C. Since the first versions of Python, this way was considered more pythonic than the first one. The third variant was introduced several years later and was widely considered more superior to the positional formatting, because it allowed to use the same value in multiple places and also allowed to assign names to placeholders. This is still the most powerful pythonic way of formatting strings. However, the most common use for string formatting in Python code is straightforward splicing of values stored elsewhere into an ad hoc composed string (as opposed to, say, storing a template with named placeholders and filling them differently on several occasions, for which `str.format()` is still the best pythonic way). This led to f-strings being introduced in Python 3.6, which are shown as the fourth variant above.

typical problems, and advise them to the developer, essentially providing powerful code completion IDE support way beyond pure syntax like bracket matching [76].

Despite some discussion on Python being the most popular programming language, as claimed by the PYPL index [10], or the third most popular one, as claimed by the TIOBE index [72], its widespead use and ever raising popularity is beyond dispute. Yet, in 2018 Alexandru et al. claimed to be the first ones to attempt forming a tangible definition and catalogue of what constitutes pythonic code. At the time of writing, we were only able to identify one other paper by Sakulniwat et al. [55] which attempts to improve upon their results. We join this initiative as well. The paper from Alexandru et al. was published along with a catalogue of idioms [34] and a repository with the idiom detection code, which makes use of the LISA library [60].

The list of idioms on Alexandru et al.'s website and paper is not complete, and there are some unavoidable reasons for that. The experiment was conducted before 2018, which coincides with the release of Python 3.7. Since then, Python 2 has also been officially deprecated [46], which forced some projects to hastily convert to Python 3. Additionally, several major Python 3 versions have been released (at the time of writing, the most recent version is 3.9.6, with a beta release of 3.10.0b4 also available). Each of these releases adds a number of features to the language [48], thus potentially enhancing the arsenal of pythonic traditions. There is obviously some adoption time for newer versions, and for these reasons, there may have been significant shifts in the popularity of idiom usage; one such idiom is seen in Figure 1. Later in the paper we will describe out process of systematically extending the catalogue of Alexandru et al.

An additional application is technical debt remediation in Python. Feltosa et al. describe the notion of *technical debt* as the result of cutting corners in the short term on the "long term sustainability" of the software project [70]. As pythonic code is considered generally more maintainable, efficient, and overall state-of-the-art, it suffices to say that being able to detect the usage of such idioms would go a long way in quantifying code quality. A potential future

application of the results of this paper could be automated detection of *anti-idioms* [80], or malpractices, in the pursuit of preventing technical debt from accumulating in the first place. A similar practice is widespread and accepted as useful in other languages, such as Java [17, 37].

Researching language traditions in general, as well as pythonic code in particular, contributes to the general body of knowledge about software languages, their design, adoption and the mutual effect languages have on programming practices and vice versa. It is a widely known fact from the software engineering lore that successful idioms from older languages tend to get "legalised" as language features in newer software languages, with the appropriate support in the form of syntactic sugar, IDE referencing and completion, static analysis checks at compile-time, correctness guarantees, etc. Researching this topic is crucial so that software languages can continue to improve and move forward. One initiative is the Software Language Engineering Body of Knowledge (*SLEBoK*) [82], which makes an effort to compare and consolidate the implementation of features and paradigms across programming and software languages. In doing so, the developers of software languages may identify discrepancies between their language and others, and then improve their own feature set.

As Shull et al. explain, replicating results of empirical studies in software engineering is key in proving their veracity, citing the difficulty of extrapolating results due to "uncontrollable sources of variation from one environment to another" [63]. The same argument holds here; the efforts of Alexandru et al. needed to be verified through an external replication, and that is one of the contributions we claim for this paper, even though we did not limit ourselves to the replication only.

In the next section we will use grounded theory [12, 25] to build the catalogue of idioms bottom-up from the available literature. Grounded theory is known to consistently deliver reliable results in many areas [35], especially for topics that attracted little prior research attention [56]. Its known disadvantages are limited generalisability (meaning that our findings might not translate well to software languages other than Python), methodological error-proneness (such as relying on one kind of data source, like interviews-only or code-only, which we explicitly cater) and the virtual impossibility to conduct a review of related work without developing assumptions [16]. As stated before, there is not much related work on researching pythonic idioms to begin with, and we follow the advice of grounded theorists on theoretical sensitivity [12, 16, 24, 25, 30], and omit the detailed review of it for the sake of methodological purity. That allows us to develop a neutral view and understanding of the chosen phenomenon of pythonicity that is neither pre-formed nor pre-theoretically developed with existing theories and paradigms [18, 64].

3 Literature Review

With the literature review, we intend to provide an answer for **RQ1**. The goal is to not only confirm the idioms that Alexandru et al. were able to identify but to further discover new pythonic idioms as well as idioms that were not covered in their research.

3.1 Suitable Sources

To discover our idioms, we use the grounded theory [12, 18, 24, 25, 30, 56] in a bottom-up approach: searching the world wide web for the most popular Python books, then scanning literature based on a set of keywords and cross-referencing the results across books. As such, we are confident that our methodology leads to uncovering all of the most commonly used pythonic idioms since the findings are rooted in a large variety of the literature available.

The literature sources were uncovered by searching the internet using key terms such as:

- *python tricks book*
- *python cookbook*
- *books "pythonic"*
- *books "idioms" "python"*

The results we found were programming blog posts, Reddit threads, StackOverflow questions, where users linked their favorite Python books. We took note of the books that were talked about the most across these sites (as well as which responses were upvoted the most) and created a list of books, articles and conferences discussing pythonic idioms.

From all the books we were able to identify, we first eliminated the complete beginner books because after reviewing them, we discovered that they focus on the fundamentals of programming in general and introducing syntax. This is not appropriate for our research, as opposed to books covering good programming practices. We also eliminated some advanced books which tend to cover Python for very specific applications and patterns, for example, data science. These are also not appropriate for our research because we want to find generalised results about the Python language as a whole rather than idioms that are only used in domain-specific applications.

The optimal balance we found was with intermediate-level books which assume that readers have prior programming knowledge of some form and generally understand the Python syntax, and read to improve their Python skills. Each book here made some reference to pythonicity, programming patterns and idioms in the description or blurb.

From the selection process, we started with the books:

- ~~*Practical Python Design Patterns: Pythonic Solutions to Common Problems* [2]~~,
- *Python Tricks: A Buffet of Awesome Python Features* [3],
- ~~*Learn Python The Hard Way* [62]~~,
- *Python Cookbook, Third Edition* [5], and
- *Effective Python: 90 Specific Ways to Write Better Python* [66].

We also reviewed several online sources, such as blog posts, which we used to confirm our previously found idioms rather than to identify new ones. We eliminated *Learn Python The Hard Way* from this list; after further review, it did not provide any useful references to pythonic idioms. Similarly, we also eliminated *Practical Python Design Patterns* because it was focused on specific use cases and design patterns rather than generalised scenarios.

Additionally, we re-reviewed a selection of 2 of the books Alexandru et al. chose:

- *Writing Idiomatic Python* [31] and
- *The Little Book of Python Anti-Patterns* [49]

This helped us to make comparisons between our newly identified idioms and the results of the original paper [1].

We scanned each source for keywords and phrases such as: "pythonic", "clean[er]", "readable", "idiom", "style", "pattern", "easy/easier", "fast", "quick", "commonly used" and "maintainable". Topics that mentioned these terms were noted down in the form of a spreadsheet, matching the topic on one axis with the sources on the other.

3.2 Identified Idioms

Having created the spreadsheet with codes, we noticed that nearly all the new idioms we managed to identify were also present in the two older sources we chose from the original paper. Conversely, almost every one of the idioms discovered in the original paper were mentioned in the newly identified literature as well. This validates the approach of the original authors, and also shows that the sources we chose were generally reliable and accurate.

We managed to find a significant amount of new idioms (29) using this approach. 4 of these idioms were filtered out due to a lack of explanation as to the use case or usefulness, being refuted as not pythonic by another conflicting source, or not being mentioned in a significant amount of sources (for example, only 1 source).

Some of the newly identified idioms, such as the "*f-strings*" feature which was released at the end of 2016 [44], were not mentioned in the older sources due to being Python features that were not widely known or used at the time of publishing; however, they have since gained attention and received mentions in our new sources. Meanwhile, the "*walrus operator*" was released with Python 3.8 [45] at the end of 2018 [32]; however, almost all of our sources were published before 2018, except for *Effective Python's Second Edition*, the only book that mentioned it. Perhaps in the future, it will gain some popularity and be discussed in newer books, but for now, we exclude it from our list.

Conversely, the "*using* `else` *after a* `for`*-loop*" idiom was discussed in the older literature sources but not in the new ones, so we also decided to filter it out. Perhaps it seemed like a good idea from the language design view but ended up being less useful in practice than anticipated.

Table 1. Overview of idioms

(a) Pythonic idiom counts

Original list of idioms	21
Newly identified idioms	29
Filtered from new list	4
Final number of new idioms	25
Total set of idioms	46
Detectable idioms from original list	21
Detectable idioms from new list	6
Total number of detectable idioms	27

(b) Full updated list of idioms [23], new detectors highlighted

```
a = b = 1
a = b if c else d
a, b = c
a, b, *c = d
a, _, c = d
a, b = b, a
@a def b():
@classmethod
@staticmethod
@property
defaultdict
namedtuple
deque
heapq
Counter
[a for b in c]
{a: b for c in d}
{a for b in c}
with
finally
dict.get()
collections.defaultdict
f'abc'
yield
next
if a: # instead of '== True'
if a is None:
if a is not None:
if not a: # instead of 'len(a) == 0'
if a in (b,c,d):
for a in b:
for a, b in enumerate(c):
for a, b in zip(c, d):
','.join(a)
def __eq__(self, other):
functools.total_ordering
PEP 8 Style Guide
assert
pprint
Naming conventions
virtualenvs
__repr__ and __str__
set(a)
for a in set(b):
a & b - c # for sets
a[b:c]
```

Having filtered out 4 idioms, we are left with 25 newly identified idioms, and together with the 21 idioms that Alexandru et al. had already covered, this comes to a total of 46 idioms covered. An overview of these numbers is given in Table 1, as well as the full list where each idiom is represented by a tiny code fragment.

3.3 The Catalogue of Idioms

After identifying these pythonic idioms, we compiled our results in the form of an online catalogue [23]. Initially, the idioms were categorised into distinct groups so that separate pages could be made for each topic. We provided definitions and explanations for each idiom, followed by simple examples of how to incorporate them in example use cases. We also provide references to a list of resources on each idiom category: links to relevant Python documentation, books that mention the topic, and where possible, links to the relevant detection code.

Interestingly, all of the identified idioms were discussed either in the Python documentation [47] or as a *PEP* (Python Enhancement Proposal) [42]. By taking these into account, as well as definitions from our chosen literature sources, we also wrote a condensed definition and purpose for each idiom. In addition, there are examples of what the "not pythonic" implementation is, which should be avoided, and provided the converse "pythonic" implementation using the idiom, taking inspiration from the Python docs and literature sources for the examples.

4 External Replication

As previously stated, one of the goals of this paper was to verify the idiom usage count results of Alexandru et al. by employing an external replication of their experiment.

Experiment 1 — replicating original results

Initially, we reached out to the authors and requested their idiom detection code which they used to produce their results. We studied the code they provided, in order to understand how it worked and observed whether there were any outdated dependencies, if the project was still able to compile, and if running the project produced any fatal run-time errors that would produce incorrect results.

Next, we replicated the experiment where Alexandru et al. ran their detector on 1,000 popular Python GitHub repositories, and observed whether or not the results were in line with what they had recorded in their paper. The replication package contained a list of the repositories that they used in the original experiment, together with the results from when the experiment was run. We re-ran the detector using the same list of repositories, with some slight differences that are discussed below.

Because the replication experiment is conducted on the latest code of each repository in the original list, some years after the original experiment, the results from this experiment will additionally help us to answer **RQ3** as we can compare the results Alexandru et al. from some time ago to new results from today. (A precise replication was impossible because the exact date(s) of the original experiments were unknown, and we could not roll back all repositories to corresponding commits closest to that day).

Discussion. After analysing their idiom detector, we can only conclude that the approach Alexandru et al. used was adequate and appropriate.

The idiom detector, written in Scala, works by pulling a Git repository using a given link, then calling a Python script that parses every Python file in the repository, making use of the built-in AST module. This results in an *abstract syntax tree*, which the detector can then analyse to count the occurrences for each idiom we are interested in by looking for patterns such as function call identifiers, keywords, or the usage of certain Python features.

The counts are accumulated per project in the form of CSV files, and the authors also include a separate Python script that can aggregate the results across all the CSVs to produce a LaTeX table.

Included in their source code was also a set of tests with sample files, where each file contained one variation of the idiom they intended to detect. We verified that these tests were appropriate and ensured that they still passed.

A limitation we identified with this approach during one of the experiments was that the detector can only find *instantiations* of certain data structures or classes, such as "`collections.namedtuple`", but not track how many times the variables are then used. This is rather difficult to detect in Python due to the lack of strong typing, and as such, there are additional uses that are not included in the results.

In the original experiment, the authors ran their detector on 997 repositories. They include the list of repositories in the form of a `.txt` file in the replication package in addition to the resulting CSV data files. However, we noticed that only 396 of the repositories in the data files overlap with the 997 sources given in the `.txt` file, which is a flaw with the replication package. We believe that sometime after the experiment, someone inadvertently re-ran the repository collection script, overwriting the original list. Nonetheless, we attempted to reconstruct the original list based on metadata from the CSV files but could not do so for 9 repositories due to incomplete metadata.

An additional issue was that 11 of the repositories used in the original experiment no longer exist. As a result, our re-run experiment had 977 repositories instead of the original 997. To counteract this, we excluded the data pertaining to the 20 missing projects from the "original" results so that we can make a meaningful comparison for the projects that were still available.

Table 2. A comparison between the results of Alexandru et al. and the reconducted experiment results (Experiment 1)
* — repaired; † — normalised
More insight needed | projects ± same, usage ± same | Projects ± same, usage up | projects, usage up

Idioms	Original results*		2021 results*		Difference†		
	Projects	Use count	Projects	Use count	Projects	Use count	
List comprehension	851	74763	848	87104	0.35%	16.51%	Group 1
Dict comprehension	143	791	194	1145	35.66%	44.75%	Group 2
Generator expression	697	32867	713	41493	2.3%	26.25%	Group 1
Decorator	753	113841	765	166569	1.59%	46.32%	Group 1
Simple magic methods	748	77999	746	81870	0.27%	4.96%	Group 0
Intermediate magic methods	412	13227	417	13007	1.21%	1.66%	Group 0
Advanced magic methods	189	2612	184	2548	2.65%	2.45%	Group 0
`finally`	496	18881	509	18887	2.62%	0.03%	Group 0
`with`	835	141435	833	219089	0.24%	54.9%	Group 1
enumerate	671	19178	676	21605	0.75%	12.66%	Group 1
`yield`	661	56396	676	56537	2.27%	0.25%	Group 0
Lambda function	653	109369	667	45632	2.14%	58.28%	Group -1
`collections.defaultdict`	310	2908	320	3996	3.23%	37.41%	Group 2
`collections.namedtuple`	258	2197	275	2589	6.59%	17.84%	Group 2
`collections.deque`	176	1685	186	1862	5.68%	10.5%	Group 2
`collections.Counter`	130	1036	158	1360	21.54%	31.27%	Group 2
`@classmethod`	512	22129	523	29615	2.15%	33.83%	Group 1
`@staticmethod`	482	11486	503	15986	4.36%	39.18%	Group 2
zip	544	14812	550	17013	1.1%	14.86%	Group 1
`itertools`	126	835	129	918	2.38%	9.94%	Group 1
`functools.total_ordering`	29	81	35	98	20.69%	20.99%	Group 2

Results. The results of this experiment can be seen in Table 2. When drawing conclusions based on our results, it is important to keep in mind that the increase in use count of idioms, can be a result of the projects themselves naturally growing over time. The most indicative metrics to consider are when the *number of projects* using a particular idiom strictly *increases* with a margin of error of 3% (7 idioms), which indicates adoption by more Python developers, or when the *use count* for an idiom strictly *decreases* (3 idioms), signaling that Python developers have begun to move away from them.

However, we also note that overall, the number of lines across all projects increased between the original experiment and the re-run by 5.67% which we can also consider as a reasonable margin of error; on average, differences larger than this indicate increased adoption as well (15 idioms).

From Table 2, we conclude that there were 5 idioms where the usage remained more or less constant, supporting the hypothesis we made. However, 15 idioms increased in popularity by looking at the increase in use count (indicated with yellow and green), thus disproving the hypothesis. Further, 7 of these idioms saw increased adoption by developers, as they were used in a significant amount of projects they previously were not (indicated in green).

Lastly, we do not consider the "`lambda`" idiom in our conclusions as we discovered that an anomalous project [68] contained an exceptionally high count (74,593) in the original data but not in the re-run (4,482), thus heavily skewing the results. We investigate this further in one of the following sections to ensure this was not due to a bug with the detector and to form a conclusion about it.

5 Beyond Replication

Previously it was discussed that one of the desired outcomes when answering **RQ2** was to provide some evidence with regards to the popularity of our newly discovered idioms, and as such, demonstrate that they are accepted as being pythonic. We can do this through Experiment 2. We also wanted to comment on how the usage of pythonic idioms had evolved to answer **RQ3**, and Experiment 3 is how we can derive these results.

Experiment 2 — detection on 1000 new repositories

Since we had previously established in § 4 that the existing detection techniques were able to accurately detect the usage of the original set of idioms, we chose 6 of the 25 new idioms we identified during the § 3 (bringing us to a total of 27 detectable idioms, as can be seen in Table 1) such that we could detect them by simply making adjustments to the existing code. Thus, we can be sure that the usage counts are accurate.

An additional reason we chose these 6 idioms is that they are built-in functions that are typically only used for one

intended purpose, and insight into the context the idioms were used is not required. One example where it would be required is when detecting the use of `set()` to clear duplicate elements in a list; there are several possible reasons to call `set()` and pinpointing the developers' intent to the purpose we are interested in is not possible through automated detection.

We subclassed some of the analyses from the original project and extended them to detect the "`heapq`", "`pprint`", "`@property`", "`__repr__` and `__str__`", "`format`" and "`join`" idioms. The new detectors can be found in a GitHub repository [19], and this repository is also referred to in our online catalogue [23]. They are not technically interesting to explain here in detail — in fact, we chose these six new idioms to detect because it was possible by making small adjustments to existing techniques in the original detection code repository. Anyone vaguely familiar with compiler technologies and tree traversals will have no trouble reading the detection code.

A fresh set of 1,000 most popular Python repositories was collected using the BASH script the original authors used, and the detector was run on this new set of repositories, using the enhanced analyses we wrote. If the usage counts of the new idioms align with those we were already able to discover, we can conclude that the newly identified idioms are demonstrably in wide use, and as such embraced as pythonic.

Results. The results from running the detector using the enhanced analyses can be found in Table 3, and the scripts, list of repositories, and result data files can be found in our GitHub repository [19].

As can be seen from the table, the inter-idiom differences in usage for Experiment 2 are aligned with the results from Experiment 1. We also observe that "`__repr__` and `__str__`" and "`@property`" are two idioms that have particularly high usage counts, being more widely used than 17 and 16 other idioms respectively. Under the assumption that the idioms Alexandru et al. discovered were pythonic, and as the usage for the new idioms are demonstrated to have similar usage statistics, we can therefore conclude that our newly identified idioms are also pythonic.

Experiment 3 — repository snapshots over time

We selected 10 repositories that were included both in the original list of repositories and our newly collected ones. The repositories we selected have been and are still under active development, which we verified by checking if there was at least one commit 6 months apart for the past 3 years, from May of 2021 going back to May of 2018. This was verified automatically using a Python script which queried the GitHub API for this information [21]. The repositories were chosen using these constraints because they are demonstrably well-maintained and receive regular updates and feature improvements, consistent with developments in Python itself. As such, they are best suited for a study on how pythonic idiom usage evolves over time.

In the same script, we collected the hashes for the first commit that landed in the chosen time periods (one commit from May and November of each year up until May 2021) for each of the 10 repositories and stored these in text files. The detector from Alexandru et al. contains an agent which takes a file containing a list of GIT repository links and automatically clones them, runs the detection, and then deletes the files afterward. We extended this agent to additionally checkout a given commit after cloning, and then run the detector. By doing this, we can essentially use snapshots of projects to detect and observe how the usage of idioms evolved every 6 months from May 2018.

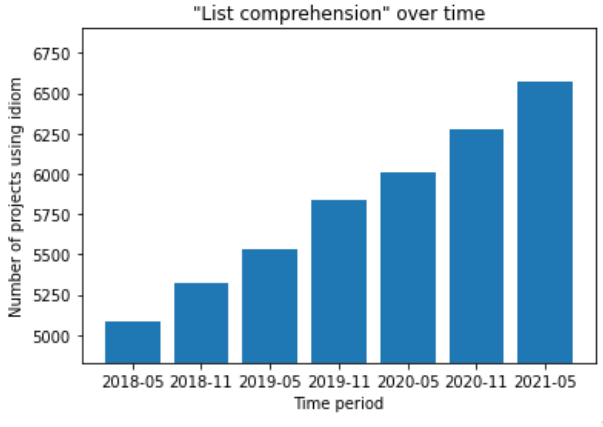
Results. Some of the resulting graphs are seen in Figure 2, and the rest can be found in the artefacts [20, 22]. We categorised the results into distinct groups in Table 4.

From this table, it is seen that the usage count against time grew over time for 21 out of the 27 detectable idioms, including 12 of the 15 idioms we previously said to have gained popularity in § 4. For the remaining idioms out of the 27, 3 idioms had usage that remained constant, and 3 had results where additional insight was needed.

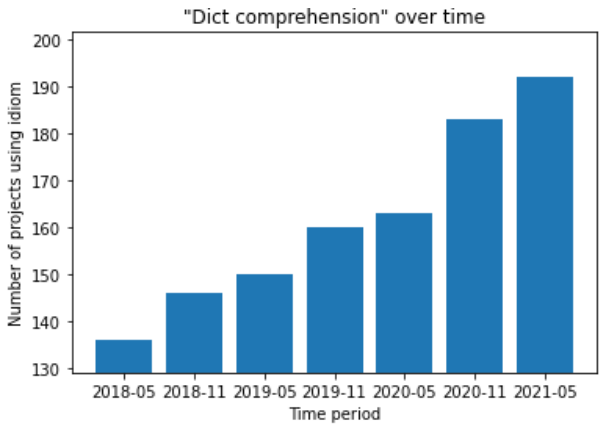
With regards to the idioms from § 4, 7 out of 15 the idioms we said to have increased in popularity saw some differences in results that are worth pointing out. In Figure 2(e), the usage of "`collections.namedtuple`" nearly doubled, then remained constant for some time. This is still an increase from where it started, however, it is worth pointing out that the usage stagnated. The same pattern is seen with "`collections.Counter`". The reason behind this is most likely the limitation we identified with the detector in § 4.

The same pattern was also seen with "`yield`", "`pprint`" and "`join`" and we believe this is due to a "saturation point" that has been reached in the projects where it was simply not necessary to use the idioms more than the already had been, or because these idioms are not as widely applicable as other idioms. The "`join`" pattern (Figure 2(c)) also showed the typical "hype curve" profile, slowly gaining popularity, then peaking and dropping to a "plateau of productivity".

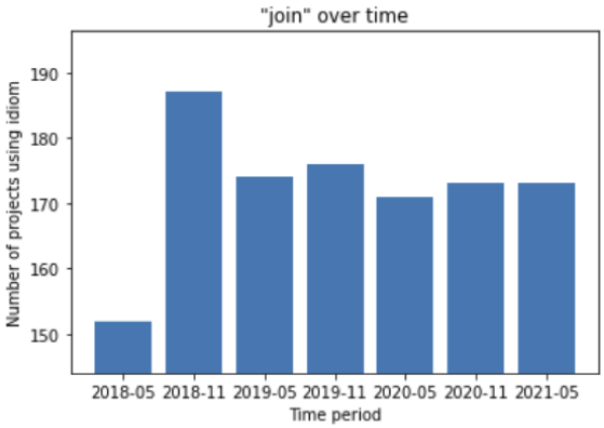
Conversely, we observed that for each of the idioms we previously claimed to have remained constant of usage, the usage actually increased for this selection of projects. Meanwhile, the usage of "`heapq`", "`functools.total_ordering`", and "`itertools`" remained more or less constant in this selection of projects. With the community context taken into account, we can explain this, as well as a visible dip between 2018 and 2020, as can be seen on Figure 2(d), by the fact that many typical functional programming instruments were forcibly, after many heated discussions, moved from the standard library to packages like `functools` and `itertools`, which in turn forced developers to update their code with references to those. Since some code did not look



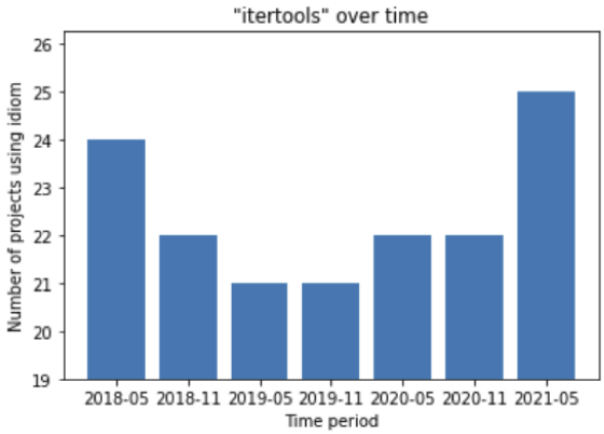
(a) list comprehension



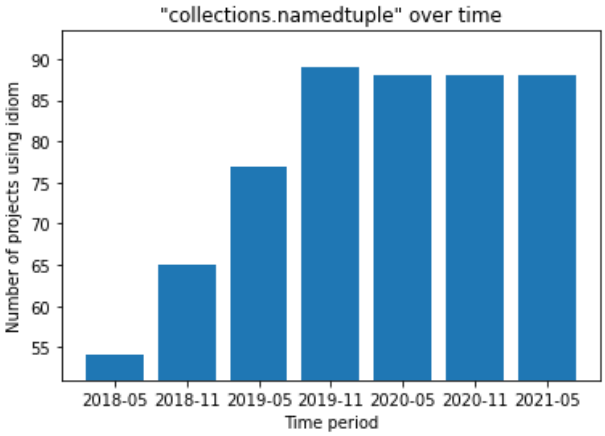
(b) dict comprehension



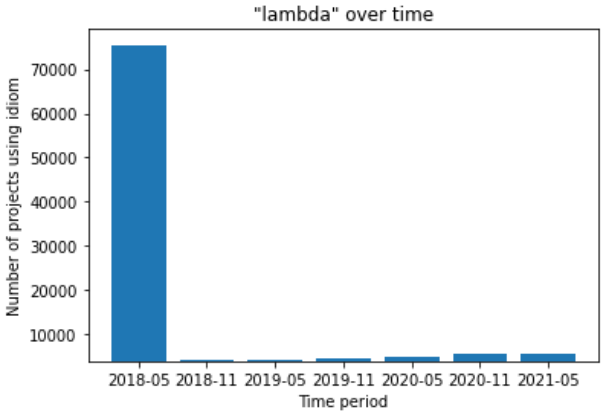
(c) join



(d) `itertools`



(e) `collections.namedtuple`



(f) `lambda` expression

Figure 2. Number of uses of some idioms over time, snapshots taken 6 months apart [22]. One can see patterns of steady linear growth **(a) (b)**, the typical full hype curve pattern **(c)**, the hype curve with the forced peak of inflated expectations **(d)**, the saturable evolution pattern **(e)** and an example of an outlier **(f)**.

Table 3. Results of Experiment 1 next to Experiment 2.

Idioms	Re-run original experiment		Experiment with new list	
	Projects	Use Count	Projects	Use Count
List comprehension	848	87104	829	56115
Dict comprehension	194	1145	144	699
Generator expression	713	41493	592	22719
Decorator	765	166569	644	101208
Simple magic methods	746	81870	650	44586
Intermediate magic methods	417	13007	263	7173
Advanced magic methods	184	2548	102	1196
finally	509	18887	325	8282
with	833	219089	872	109501
enumerate	676	21605	705	18071
yield	676	56537	518	35768
lambda	667	45632	557	23498
collections.defaultdict	320	3996	258	2589
collections.namedtuple	275	2589	200	1472
collections.deque	186	1862	126	697
heapq	—	—	43	193
collections.Counter	158	1360	129	840
@classmethod	523	29615	380	16711
@staticmethod	503	15986	435	11841
@property	—	—	464	37795
zip	550	17013	552	12553
itertools	129	918	74	445
functools.total_ordering	35	98	21	58
__repr__ and __str__	—	—	470	15031
pprint	—	—	131	1076
format	—	—	165	2720
join	—	—	143	4617

all that pythonic anymore after the migration, developers were refactoring their code by either decreasing the use of `functools` and `itertools` functionality, or encapsulating it within a wrapper library of their own, thus in any case decreasing the observable use of the concerned idioms. After that transition period, the usual expected growth continued naturally with the growth of the lines of code in a codebase. The authors anecdotally confirm recalling going through the same process in their own codebases.

While these results are noteworthy, they only pertain to a small sample of projects, and the larger sample of 977 projects shows more generalised results across the Python community.

We also observed once again that in Figure 2(f), as with the results of Experiment 1, the usage of the anomalous "`lambda`" idiom decreased from roughly 75,347 uses to 3,940. This is due to the same anomalous project from the § 4 being present in our selection of repositories here. To investigate this further, we cloned the repository using the commit hash from the original experiment's metadata and ran a `grep` search through the repository. By doing so, we verified that the exceptionally high use count of "`lambda`" was legitimate and not due to a bug with the detector. We then cloned the repository from the next timeframe (November 2018) and ran the search again, and indeed the usage of "`lambda`" was significantly lower, caused by a major refactoring effort.

The conclusion here is that while the numbers in Table 2 were correct, the choice of including this repo in Alexandru et al.'s experiment heavily skewed the popularity of "`lambda`" in the original results. However, after the anomalous time period, the usage then rose again steadily to 5,498 as seen in Figure 2(b), we can conclude here that "`lambda`" grew in popularity for this selection of repositories.

A similar result was observed with `@staticmethod`, which increased in usage since May of 2018, but curiously, the usage decreased by nearly a quarter between November 2020 and May 2021. After investigating these two snapshots, we observed a similar result as with "`lambda`" — once again, the same anomalous project underwent another refactor which caused the usage of "`@staticmethod`" to decrease from 515 to 314, skewing the results again. Nonetheless, we again conclude that "`@staticmethod`" grew in popularity by referring to our generalised results from Table 2.

Table 4. Results of Experiment 3, categorised.
More insight needed | usage ± constant | usage up, then stagnated | usage up

Idioms	Conclusions
List comprehension	Group 2
Dict comprehension	Group 2
Generator expression	Group 2
Decorator	Group 2
Simple magic methods	Group 2
Intermediate magic methods	Group 2
Advanced magic methods	Group 2
finally	Group 2
with	Group 2
enumerate	Group 2
yield	Group 1
lambda	Group -1
collections.defaultdict	Group 2
collections.namedtuple	Group 1
collections.deque	Group 2
heapq	Group 0
collections.Counter	Group 1
@classmethod	Group 2
@staticmethod	Group -1
@property	Group 2
zip	Group 2
itertools	Group 0
functools.total_ordering	Group 0
__repr__ and __str__	Group 2
pprint	Group 1
format	Group -1
join	Group 1

As for the 6 newly identified idioms we were able to write detectors for, "`heapq`" and "`pprint`" stay constant in terms of usage, while "`@property`" went up. The usage of "`format`" decreased, and we hypothesise that this is due to the introduction of "f-strings", one of the idioms we covered that was introduced in Python 3.6 [69], and as we previously illustrated, a cleaner way to approach string interpolation. Additionally, the usage of "`join`" increased then stayed constant, and "`__repr__` and `__str__`" increased steadily.

6 Conclusion

Through the course of this paper, using the great efforts of Alexandru et al. as a foundation, we dove deeper into the ecosystem of Python and its pythonic values. In this final section of the paper, we provide definitive answers for our research questions through the results we gathered across each of our experiments and the literature review.

6.1 Results

Research Question 1. Through a literature review performed in the principles of grounded theory, we were able to uncover 25 new pythonic idioms, increasing the total number of pythonic idioms discovered to 46. An overview of the idiom counts and their list is seen in Table 1. Every idiom is given with detailed definitions, use cases, links to detectors, and examples inspired by the literature sources, and these can be found on our online catalogue [23]. Further, the data files, list of repositories, aggregation scripts, and detectors can be found in a GitHub repository [19].

Research Question 2. We extended Alexandru et al.'s detectors to include 6 out of the 25 newly identified idioms. From the results of our experimentation (Experiments 2 and 3), we have established that each of the 6 idioms are under wide use in the most popular Python projects, and as such, are pythonic. The usage statistics are comparable to the idioms previously identified by Alexandru et al. as pythonic; in some cases, even more so, by observing the number of projects using the "`__repr__` and `__str__`" and "`@property`" idioms in Table 3.

Research Question 3. As can be seen in Table 1, we were able to experiment on a set of 27 idioms, of which 21 were from the original set of idioms, and 6 were part of our newly identified idioms.

6.2 Threats to Validity

The biggest threat to validity in our project comes from using the number of times a particular pattern occurs in the code, as a proxy for that code being pythonic. The essence of the *Zen of Python* [43] is in principles like "beautiful is better than ugly" and "should be one obvious way to do it", which are principles of software design, inherently inexpressible in automatic detectors. The technical subject of this study is indeed the number of times a particular idiom occurs in code (over time), and not how those statistics correspond to the actual perceived pythonicity of the code by an expert. This is particularly important in the context of idiom evolution: for example, a codebase that moves away from named tuples because it grows and needs actual fully implemented classes, does not necessarily lose in pythonicity.

Python is also a dynamically typed language with a lot of features that make it hard to parse and analyse [75]. Hence, there is a good chance that the detectors suffer from false negatives or positives. Yet, we expect few problems stemming from deliberate abuse of the language: as we know from prior research on Python [38], as well as on PHP [29] and on JavaScript [52], most harmful language constructs tend to be avoided by developers in practice or used in a harmless way.

Just like we have claimed some degree of incompleteness in the work of Alexandru et al [1] because their experiments were completed before the release of Python 3.7, our own

replication and extension was performed before September 2021 when Python 3.10 was released, incling new features like natural syntax for union types and structural pattern matching with the `match` statement. With time, some of these new features have a chance of becoming new pythonic idioms worthy of adding to our catalogue.

With regards to the internal validity of our literature review to discover idioms, as our literature sources [3, 5, 31, 49, 66] were chosen based on sentiment from posters on online media such as forums and blogs, it is possible that the most prominent results are purported by highly opinionated individuals, not representative of the majority of the Python community. We also have assumed that the idioms that are discussed in literature sources and idioms that are used in actual code, overlap heavily. Finally, it is probable that idioms authors choose to write out, are biased to their own personal preferences, which means that certain idioms are given more attention than others, or that there is a conflict of opinions among sources. We aimed to alleviate most of these threats to validity by cross-referencing our results across a large variety of different sources, and through our filtering process, we hope to have ruled out most instances of bias.

A threat to the external validity of this research could arise from the choice of only using prominent open source projects during our idiom detection phase, which threatens the generalisability of our results to the pythonicity of closed source projects. There is some debate about the code quality of open source projects as opposed to proprietary or closed source projects. As Raghunathan et al. contend, open source software projects often suffer from the "free rider effect" where the majority of users prefer to benefit from the efforts of the few who actually contribute to the projects [50]. Proponents in favour of closed source projects would claim that as developers have a monetary incentive (through employment), the quality of contributions to closed source code is higher. Conversely, it has been experimentally demonstrated that providing extrinsic motivation to workers through rewards (e.g., paying them a salary) conflicts with intrinsic motivation [7]. In the short-term, extrinsically motivated workers are more interested in the work than intrinsically motivated workers (those who simply enjoy the work, without any rewards), but even after removing the rewards, those who were already working without rewards will be more interested in the work than those who have now lost the rewards. In other words, in the long term, monetary incentives harm the quality of code. In cognitive psychology and economics this phenomenon is known as overjustification [71]. Currently it can be assumed that the quality of code in closed source projects is not necessarily higher than that of open source projects [50]; as such, we believe that our results are generalisable to closed source projects as well.

6.3 Future Work

A wide array of research based on this paper can be conducted, some very straightforward such as writing new detectors for the remaining new idioms, as well as discovering their usage in projects. The anomalous project [68] that we have pinpointed (and excluded) here, could also be examined closer. A less trivial application, as we previously described, would also be to determine and detect all of the "anti-patterns" in Python, for which we illustrated potential uses in the introductory sections.

There is a certain connection between the concept of pythonicity and the results we have obtained as a part of this extended replication — or so we assumed. However, the nature of this connection, its inevitability, proportion and persistence, can and should be investigated further. For example, this can be done by comparing detector output with human assessment, or by specifically looking for code fragments where local application of pythonic idioms led to unpythonic designs.

6.4 Lessons Learnt

To conclude, we have investigated the notion of pythonicity by replicating a previous research effort by Alexandru et al. [1], strengthening the statements made in the original paper by generalising them to a wider collection of Python idioms [23], applying the detection procedure with a wider collection of detectors [19] to a wider selection of open source repositories, using sources of different kind to collect ground data, and finally taking first steps in investigating how the usage of pythonic idioms evolves over time. The linked artefact is released publicly [22].

We uncovered from the results in Experiment 2 and Experiment 3 that from the original list of pythonic idioms provided by Alexandru et al., 16 out of 21 idioms saw an increase in popularity, including the "`lambda`" idiom which required additional research through Experiment 3. 5 idioms' usage remained constant (all 3 of the "magic method" categories, as well as "`finally`" and "`yield`"). In addition to the original set of idioms, 2 of the 6 new idioms we wrote detectors for, had constant usage ("`heapq`" and "`pprint`"), 3 saw increased usage ("`@property`", "`join`", "`__repr__` and `__str__`") and the usage of "`format`" decreased. For the latter we conjectured potential reasons, which in turn now require additional research to be validated.

We hope this replication of the research on coding traditions in Python, augmented by the analysis of the evolution of these coding traditions, will serve as a good stepping stone to future research in this area.

References

[1] Carol V. Alexandru, José J. Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. 2018. On the Usage of Pythonic Idioms. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 1–11. https://doi.org/10.1145/3276954.3276960

[2] Wessel Badenhorst. 2017. *Practical Python Design Patterns: Pythonic Solutions to Common Problems*. Apress. https://www.apress.com/gp/book/9781484226797

[3] Dan Bader. 2017. *Python Tricks: A Buffet of Awesome Python Features*. Dan Bader. https://realpython.com/products/python-tricks-book/

[4] Mark W. Bailey and Jack W. Davidson. 1995. A Formal Model and Specification Language for Procedure Calling Conventions. In *Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL)*. ACM, 298–310. https://doi.org/10.1145/199448.199517

[5] David Beazley and Brian K. Jones. 2013. *Python Cookbook* (3rd ed.). O'Reilly Media, Inc.

[6] Dmitrij Yu. Boulychev, Dmitrij V. Koznov, and Andrey A. Terekhov. 2002. On Project-Specific Languages and Their Application in Reengineering. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 177–185. https://doi.org/10.1109/CSMR.2002.995802

[7] Roland Bénabou and Jean Tirole. 2003. Intrinsic and Extrinsic Motivation. *The Review of Economic Studies* 70, 3 (07 2003), 489–520. https://doi.org/10.1111/1467-937X.00253

[8] Robert D. Cameron. 2002. Four Concepts in Programming Language Description: Syntax, Semantics, Pragmatics and Metalanguage. https://www2.cs.sfu.ca/~cameron/Teaching/383/syn-sem-prag-meta.html.

[9] Uriel Campos, Guilherme Smethurst, João Pedro Moraes, Rodrigo Bonifácio, and Gustavo Pinto. 2019. Mining Rule Violations in JavaScript Code Snippets. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*. IEEE / ACM, 195–199. https://doi.org/10.1109/MSR.2019.00039

[10] Pierre Carbonnelle. 2021. PopularitY of Programming Language. Online: https://pypl.github.io/PYPL.html.

[11] James Coplien. 1997. Advanced C++ Programming Styles and Idioms. In *Proceedings of the 25th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE Computer Society, 352. https://doi.org/10.1109/TOOLS.1997.681881

[12] Juliet M. Corbin and Anselm Strauss. 2014. *Basics of Qualitative Research. Techniques and Procedures for Developing Grounded Theory* (4th ed.). Sage.

[13] Vittorio Cortellessa, Antinisca Di Marco, Romina Eramo, Alfonso Pierantonio, and Catia Trubiani. 2010. Digging into UML Models to Remove Performance Antipatterns. In *Proceedings of the Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (QUOVADIS)*. ACM, 9–16. https://doi.org/10.1145/1808877.1808880

[14] Dario Di Nucci, Hoang Son Pham, Johan Fabry, Coen De Roover, Kim Mens, Tim Molderez, Siegfried Nijssen, and Vadim Zaytsev. 2019. A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns. In *Post-proceedings of the 12th Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE) (CEUR Workshop Proceedings, Vol. 2510)*, Anne Etien (Ed.). CEUR-WS.org, 38–44. http://ceur-ws.org/Vol-2510/sattose2019_paper_3.pdf

[15] Shouki A. Ebad and Danish Manzoor. 2016. An Empirical Comparison of Java and C# Programs in Following Naming Conventions. *International Journal of People-Oriented Programming (IJPOP)* 5, 1 (2016), 39–60. https://doi.org/10.4018/IJPOP.2016010103

[16] Mohamed El Hussein, Sandra Hirst, Vince Salyers, and Joseph Osuji. 2014. Using Grounded Theory as a Method of Inquiry: Advantages and Disadvantages. *The Qualitative Report* 19, 27 (2014), 1–15.

[17] Eva van Emden and Leon Moonen. 2002. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 97–106. https://doi.org/10.1109/WCRE.2002.1173068

[18] H. Engward. 2013. Understanding grounded theory. *Nursing Standard* 28, 7 (2013), 37–41. https://doi.org/10.7748/ns2013.10.28.7.37.e7806

[19] Aamir Farooq. 2021. Detect Your Zen: Experimentation and Detection Code. https://github.com/SlimShadyIAm/DetectYourZen.

[20] Aamir Farooq. 2021. *How To Zen Your Python – Graphs*. Technical Report. University of Twente. https://doi.org/10.6084/m9.figshare.14782170.v1.

[21] Aamir Farooq. 2021. `get-commits.py`: Python commit fetching script. https://github.com/SlimShadyIAm/DetectYourZen/blob/main/src/main/python/get-commits.py.

[22] Aamir Farooq and Vadim Zaytsev. 2021. *Artefact for "There Is More Than One Way to Zen Your Python"*. Technical Report. University of Twente. https://doi.org/10.6084/m9.figshare.16825933.

[23] Aamir Farooq and Vadim Zaytsev. 2021. Zen Your Python. https://slimshadyiam.github.io/ZenYourPython.

[24] Barney G. Glaser. 1998. *Doing Grounded Theory: Issues and Discussions*. Vol. 254. Sociology Press.

[25] Barney G. Glaser and Anselm L. Strauss. 2017. *Discovery of Grounded Theory: Strategies for Qualitative Research*. Routledge.

[26] Boryana Goncharenko and Vadim Zaytsev. 2016. Language Design and Implementation for the Domain of Coding Conventions. In *Proceedings of the International Conference on Software Language Engineering (SLE)*. ACM, 90–104. https://doi.org/10.1145/2997364.2997386

[27] DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixão, and Giovanni Rosa. 2020. Does Code Review Really Remove Coding Convention Violations?. In *Proceedings of the 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 43–53. https://doi.org/10.1109/SCAM51674.2020.00010

[28] Michael P. Heinl, Alexander Giehl, and Lukas Graif. 2020. AntiPatterns Regarding the Application of Cryptographic Primitives by the Example of Ransomware. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES)*. ACM, 64:1–64:10. https://doi.org/10.1145/3407023.3409182

[29] Mark Hills. 2015. Evolution of Dynamic Feature Usage in PHP. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 525–529. https://doi.org/10.1109/SANER.2015.7081870

[30] Margaret H. Kearny. 2001. New Directions in Grounded Formal Theory. In *Using Grounded Theory in Nursing*. Springer, 227–246.

[31] Jeff Knupp. 2013. *Writing Idiomatic Python 3.3*. Createspace Independent Pub.

[32] Łukasz Langa. 2018. PEP 569 – Python 3.8 Release Schedule. https://www.python.org/dev/peps/pep-0569/.

[33] Cristian Mateos, Alejandro Zunino, Andres Flores, and Sanjay Misra. 2019. COBOL Systems Migration to SOA: Assessing Antipatterns and Complexity. *Inf. Technol. Control.* 48, 1 (2019), 71–89. https://doi.org/10.5755/j01.itc.48.1.21566

[34] Jose Javier Merchante. 2019. Pythonic Examples. https://pythonic-examples.github.io/.

[35] P. Jane Milliken and Rita Schreiber. 2012. Examining the Nexus between Grounded Theory and Symbolic Interactionism. *International Journal of Qualitative Methods* 11, 5 (2012), 684–696. https://doi.org/10.1177/160940691201100510

[36] Markus Mohnen. 1996. Context Patterns in Haskell. In *Proceedings of the Eighth International Workshop on Implementation of Functional Languages (IFL) (LNCS, Vol. 1268)*. Springer, 41–57. https://doi.org/10.1007/3-540-63237-9_18

[37] M. J. Munro. 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of the International Software Metrics Symposium*, Vol. 2005. IEEE, 15–24. https://doi.org/10.1109/METRICS.2005.3

[38] Bence Nagy, Tibor Brunner, and Zoltán Porkoláb. 2020. Unambiguity of Python Language Elements for Static Analysis. In *Proceedings of the 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 1–12. https://doi.org/10.1109/SCAM52516.2021.00017

[39] Takao Okubo and Hidehiko Tanaka. 2007. Secure Software Development through Coding Conventions and Frameworks. In *Proceedings of the Second International Conference on Availability, Reliability and Security (ARES)*. IEEE Computer Society, 1042–1051. https://doi.org/10.1109/ARES.2007.131

[40] Aggelos Papamichail, Apostolos V. Zarras, and Panos Vassiliadis. 2020. Do People Use Naming Conventions in SQL Programming?. In *Proceedings of the 46th International Conference on Current Trends in Theory and Practice of Computer Science (LNCS, Vol. 12011)*. Springer, 429–440. https://doi.org/10.1007/978-3-030-38919-2_35

[41] Howard A. Peell. 1987. An APL idiom inventory. In *Proceedings of the International Conference on APL: APL in Transition*. ACM, 362–368. https://doi.org/10.1145/28315.28360

[42] Python Software Foundation. 2000. PEP 0 – Index of Python Enhancement Proposals (PEPs). https://www.python.org/dev/peps/.

[43] Python Software Foundation. 2004. The Zen Of Python. https://github.com/python/peps/blob/master/pep-0020.txt.

[44] Python Software Foundation. 2016. What's New In Python 3.6. https://docs.python.org/3/whatsnew/3.6.html.

[45] Python Software Foundation. 2019. What's New In Python 3.8. https://docs.python.org/3/whatsnew/3.8.html.

[46] Python Software Foundation. 2020. Sunsetting Python 2. https://www.python.org/doc/sunset-python-2/.

[47] Python Software Foundation. 2021. Python 3.9.7 documentation. https://docs.python.org/3/.

[48] Python Software Foundation. 2021. What's New In Python. https://docs.python.org/3/whatsnew/index.html.

[49] Quantified Code. 2014. The Little Book of Python Anti-Patterns. https://github.com/quantifiedcode/python-anti-patterns.

[50] Srinivasan Raghunathan, Ashutosh Prasad, Birendra Mishra, and Hsihui Chang. 2005. Open Source Versus Closed Source: Software Quality in Monopoly and Competitive Markets. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* 35 (12 2005), 903 – 918. https://doi.org/10.1109/TSMCA.2005.853493

[51] K. Reitz and T. Schlusser. 2016. *The Hitchhiker's Guide to Python: Best Practices for Development*. O'Reilly Media.

[52] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do — A Large-Scale Study of the Use of Eval in JavaScript Applications. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 6813)*. Springer, 52–78. https://doi.org/10.1007/978-3-642-22655-7_4

[53] Catherine Roussey, Óscar Corcho, and Luis Manuel Vilches Blázquez. 2009. A Catalogue of OWL Ontology Antipatterns. In *Proceedings of the Fifth International Conference on Knowledge Capture (K-CAP)*. ACM, 205–206. https://doi.org/10.1145/1597735.1597784

[54] James W. Rymarczyk. 1982. Coding Guidelines for Pipelined Processors. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, 12–19. https://doi.org/10.1145/800050.801821

[55] Tattiya Sakulniwat, Raula Gaikovina Kula, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Dong Wang, Takashi Ishio, and Kenichi Matsumoto. 2019. Visualizing the Usage of Pythonic Idioms Over Time: A Case Study of the `with open` Idiom. In *Proceedings of the 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 43–435. https://doi.org/10.1109/IWESEP49350.2019.00016

[56] N. J. Salkind. 2010. Grounded theory. *Encyclopedia of research design* 1 (2010), 549–553. https://doi.org/10.4135/9781412961288.n169

[57] Kevin Schneider and Timo Mühlhaus. 2019. FSharpGephiStreamer: An Idiomatic Bridge between F# and Network Visualization. *J. Open Source Softw.* 4, 38 (2019), 1445. https://doi.org/10.21105/joss.01445

[58] Dale Schumacher. 2012. Actor Idioms. In *Proceedings of the Second Workshop on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions (AGERE!)*. ACM, 123–128. https://doi.org/10.1145/2414639.2414655

[59] Michael L. Scott. 2021. *Programming Language Pragmatics, Third Edition* (4th ed.). Morgan Kaufmann, San Francisco, CA, USA.

[60] seal UZH. 2019. LISA Workspage. https://bitbucket.org/sealuzh/workspace/projects/LISA.

[61] Vibhu Saujanya Sharma and Samit Anwer. 2014. Performance Antipatterns: Detection and Evaluation of Their Effects in the Cloud. In *Proceedings of the International Conference on Services Computing (SCC)*. IEEE Computer Society, 758–765. https://doi.org/10.1109/SCC.2014.103

[62] Zed A. Shaw. 2017. *Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code* (1st ed.). Addison-Wesley Professional.

[63] F. Shull, V. Basili, J. Carver, J.C. Maldonado, G.H. Travassos, M. Mendonca, and S. Fabbri. 2002. Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem. In *Proceedings International Symposium on Empirical Software Engineering*. IEEE, 7–16. https://doi.org/10.1109/ISESE.2002.1166920

[64] Odis E. Simmons. 2006. Some Professional and Personal Notes on Research Methods, Systems Theory, and Grounded Action. *World Futures* 62, 7 (2006), 481–490. https://doi.org/10.1080/02604020600912772

[65] Brett Slatkin. 2015. *Effective Python: 59 Specific Ways to Write Better Python* (1st ed.). Addison-Wesley Professional.

[66] Brett Slatkin. 2019. *Effective Python: 90 Specific Ways to Write Better Python, 2nd Edition*. Addison-Wesley Professional.

[67] Michael Smit. 2019. Code Convention Adherence in Research Data Infrastructure Software: An Exploratory Study. In *Proceedings of the International Conference on Big Data*. IEEE, 4691–4700. https://doi.org/10.1109/BigData47090.2019.9006130

[68] Christopher Smith, Aaron Meurer, Mateusz Paprocki, Oscar Benjamin, Matthew Rocklin, S. Y. Lee, Ondřej Čertík, Francesco Bonazzi, Oscar Gustafsson, Julien Rioux, et al. 2007–2021. SymPy: Python Library for Symbolic Mathematics. https://github.com/sympy/sympy.

[69] Eric V. Smith. 2015. PEP 498 – Literal String Interpolation. https://www.python.org/dev/peps/pep-0498/.

[70] Jie Tan, Daniel Feitosa, Paris Avgeriou, and Mircea Lungu. 2021. Evolution of Technical Debt Remediation in Python: A Case Study on the Apache Software Ecosystem. *Journal of Software: Evolution and Process* 33, 4 (2021), e2319. https://doi.org/10.1002/smr.2319

[71] Shu-Hua Tang and Vernon C. Hall. 1995. The overjustification effect: A meta-analysis. *Applied Cognitive Psychology* 9, 5 (1995), 365–404. https://doi.org/10.1002/acp.2350090502

[72] TIOBE. 2021. TIOBE Index: C, Java, and Python compete for the first position. https://www.tiobe.com/tiobe-index/.

[73] Federico Tomassetti and Vadim Zaytsev. 2020. Reflections on the Lack of Adoption of Domain Specific Languages. In *STAF Workshop Proceedings (STAF, Vol. 2707)*. CEUR-WS.org, 85–94. http://ceur-ws.org/Vol-2707/oopslepaper5.pdf

[74] F. Turbak, D. Gifford, and M. A. Sheldon. 2008. *Design Concepts in Programming Languages*. MIT Press.

[75] 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

[76] Indika P. Wickramasinghe and Harsha K. Kalutarage. 2021. Naive Bayes: applications, variations and vulnerabilities: a review of literature with code snippets for implementation. *Soft Comput.* 25, 3 (2021), 2277–2293. https://doi.org/10.1007/s00500-020-05297-6

[77] Eliane Stampfer Wiese, Anna N. Rafferty, Daniel M. Kopta, and Jacqulyn M. Anderson. 2019. Replicating Novices' Struggles with Coding Style. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*. IEEE / ACM, 13–18. https://doi.org/10.1109/ICPC.2019.00015

[78] Eliane S. Wiese, Michael Yen, Antares Chen, Lucas A. Santos, and Armando Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In *Proceedings of the Fourth ACM Conference on Learning at Scale (L@S)*. ACM, 41–50. https://doi.org/10.1145/3051457.3051469

[79] Samim Yaquby. 2019. Syntax, semantics, and pragmatics. https://samimyaquby.medium.com/syntax-semantics-and-pragmatics-14939488d1c9.

[80] Moshe Zadka. 2015. Idioms and Anti-Idioms in Python. http://omz-software.com/-editorial/docs/howto/doanddont.html.

[81] Vadim Zaytsev. 2013. Micropatterns in Grammars. In *Proceedings of the Sixth International Conference on Software Language Engineering (LNCS, Vol. 8225)*. Springer, 117–136. https://doi.org/10.1007/978-3-319-02654-1_7

[82] Zaytsev, V. (Ed.). 2009–2021. Software Language Engineering Body of Knowledge. http://slebok.github.io.