On the Usage of Pythonic Idioms

Carol V. Alexandru
University of Zurich
Switzerland
alexandru@ifi.uzh.ch

José J. Merchante
Universidad Rey Juan Carlos
Spain
jj.merchante@alumnos.urjc.es

Sebastiano Panichella
University of Zurich &
Zurich University of Applied Sciences
Switzerland
panichella@ifi.uzh.ch

Sebastian Proksch
University of Zurich
Switzerland
proksch@ifi.uzh.ch

Harald C. Gall
University of Zurich
Switzerland
gall@ifi.uzh.ch

Gregorio Robles
Universidad Rey Juan Carlos
Spain
grex@gsyc.urjc.es

Abstract

Developers discuss software architecture and concrete source code implementations on a regular basis, be it on question-answering sites, online chats, mailing lists or face to face. In many cases, there is more than one way of solving a programming task. Which way is best may be decided based on case-specific circumstances and constraints, but also based on convention. Having strong conventions, and a common vocabulary to express them, simplifies communication and strengthens common understanding of software development problems and their solutions. While many programming ecosystems have a common vocabulary, Python's relationship to conventions and common language is a particularly pronounced. The "Zen of Python", a famous set of high-level coding conventions authored by Tim Peters, states "There should be one, and preferably only one, obvious way to do it". This 'one way to do it' is often referred to as the 'Pythonic' way: the ideal solution to a particular problem. Few other programming languages have coined a unique term to label the quality of craftsmanship gone into a software artifact. In this paper, we explore how Python developers understand the term 'Pythonic' by means of structured interviews, build a catalogue of 'pythonic idioms' gathered from literature, and conjecture on the effects of having a language-specific term for quality code, considering the potential it could hold for other programming languages and ecosystems. We find that while the term means different things to novice versus experienced Python developers, it encompasses not only concrete implementation, but a way of thinking – a culture – in general.

CCS Concepts • **Software and its engineering → Patterns**; *Multiparadigm languages*; *Scripting languages*; *Designing software*;

Keywords Python, Pythonic, Conventions, Programming, Idioms, Culture, Community

ACM Reference Format:
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 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3276954.3276960

1 Introduction

Bloch [10] describes a programming language as a complex ecosystem that consists of three parts: syntax, vocabulary and effective ways to use the language in actual projects. The former two are provided by the language designer from the inception of a language, however the third emerges from the community over time and is partially out of their control. How a language ecosystem evolves not only depends on the inherent qualities of the language (e.g., whether it is easy to learn or suitable for high-performance applications) but also on how parts of the community interact (e.g., veterans and novices or professionals and amateurs). Key to this interaction is a common language and an agreement on programming concepts and software construction principles. Besides concepts and terms provided by the language designer (for example 'Monads' in Haskell or 'Closures' in languages like JavaScript), the community develops an understanding of idioms and anti-idioms which may phase in and out of fashion over time. For example, widely respected 'design patterns' for Java were presented by Grand [18] who were not themselves Java language designers. Though over time, by means of trial and error, the community further evolves the language and certain concepts can also fall out of

Onward! '18, November 7–8, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6031-9/18/11…$15.00
https://doi.org/10.1145/3276954.3276960

favor. For example, the 'Singleton' design pattern has been described as dangerous [12] and use of the 'Singleton' is now widely discouraged, regardless of whether it is inherently bad, or whether it has simply been misunderstood and over-used by inexperienced programmers.

In general, developing a common vocabulary, such as the 'design patterns' for Java, enables effective discussions on architecture and implementation, and many languages employ such a vocabulary to a certain degree. We postulate, however, that Python is special: the community has a word, 'Pythonic', to describe almost anything positive relating to implementation and architecture. For example, someone may state that "It would be more pythonic to write it this way..." or "The pythonic way would be to...". As such, the term 'Pythonic' is understood to be an *idiomatic way* of writing Python. It describes a common understanding of what 'good' Python code is, even when no formal specification exists. The importance of this concept to the community can be seen online: on StackOverflow, ~806k out of ~896k or 90% of threads tagged with 'Python' mention the word 'Pythonic'.

Over many years, the Python programming language has been gaining more and more traction and is now one of the most popular languages. Following tremendous growth over the past few years, which is reflected in the increasing number of discussions on Stack Overflow[1], Python ranks numbers 4 in the Tiobe index[2] and GitHub hosts over 2 million repositories containing Python code (not counting forks)[3]. As an approachable, general purpose programming language, Python is popular as a first programming language to learn and it is widely used by professionals in other disciplines, such as natural sciences, machine learning and data analysis in general as an alternative to languages such as R or Matlab.

In this paper, we perform groundwork towards understanding the concept of a language-specific 'quality brand' which seems to be particularly pronounced in the Python community. The term 'Pythonic' has been with the Python community for over a decade, yet it has never been thoroughly researched. Thus, we investigate what it means to Python developers, how pythonic idioms are learned and how widespread their usage is in actual source code. We then consider the implications of this term on the development and spread of the language. More specifically, we investigate the following research questions:

RQ$_1$ *Is 'Pythonic' a known concept? Is it desirable? Why?* We investigate if 'Pythonic' is a widely known and accepted term among Python developers. We want to find out whether developers strive to write pythonic code and if they agree with the ideas and goals it reportedly represents. We also want to understand how developers learn about the concept of pythonic code, and how the understanding of pythonic ideas spreads in the community.

RQ$_2$ *What are concrete pythonic idioms? How widespread are they?* Given that 'Pythonic' appears to encompass a certain way of writing source code, we want to catalog the idioms developers describe as being pythonic. We also want to observe how frequently these idioms are used in real-world projects.

To answer these questions, we use a mixed-method approach. First, we conducted a series of interviews with both novice and experienced Python developers to learn about their understanding and opinion on the term 'Pythonic'. Second, we analyzed 1,000 projects to observe how common different pythonic idioms are in existing source code.

Based on our findings, we consider the potential impact on the community and on professional developers. We hypothesize that applying pythonic practices may play an important role in communicating developers skills and expertise. For individual developers, striving to write more pythonic source code may eventually award higher status within the community. We also want to understand whether being knowledgeable about pythonic code represents an advantage at work, for example when doing job interviews. In broader terms, we want to know the effect of having a dedicated word to describe *the way* within a programming ecosystem.

We find that the term 'Pythonic' provides a cultural anchor, akin to a *brand*, for community members to signal both skills desirable in a developer and concrete properties desirable in written source code. We postulate that having this term simplifies curating a common understanding of "good coding", even if the term itself has no clear-cut definition.

To summarize, this paper presents the following main contributions:

- We are the first to study the term 'Pythonic' and its prevalence in the Python community.
- We find that Python developers are indeed very aware of the term 'Pythonic' in general and that the awareness and interpretation of the term evolves with developer experience.
- We learn that, while encompassing concrete idioms, 'Pythonic' goes further and refers to a shared philosophy and culture.
- We create a catalog of pythonic idioms and confirm their adoption in open-source projects.
- We interpret the effects of a dedicated quality term and provide a vision of how other ecosystems could benefit from having such an 'idiomatic brand'.

2 Background

In this section we first introduce related work concerning idiomatic source code and then contrast this general picture with the specifics of Python highlighting the peculiar and distinct characteristics of the Python community.

[1] https://stackoverflow.blog/2017/09/06/incredible-growth-python/
[2] https://www.tiobe.com/tiobe-index/
[3] https://github.com/search?q=language%3APython&type=Repositories

2.1 Related Work

The general idea of writing idiomatic code is not a new concept. Perlis and Rugaber [28] postulated in 1979 that advanced programming needs to go beyond syntax and into convention and style as a means to guide the structure of a program. Style and idioms can be used to express design concepts and reusable abstractions.

Later, Gil and Lorenz [17] found that a fundamental difference exists between design patterns and language design. They establish a taxonomy and define *patterns* as universally reusable concepts, independent of an implementation. While very foundational patterns are implemented as *features* of programming languages (e.g., inheritance), other foundational patterns exist that are not language features (e.g., the 'visitor' pattern). Instead, they conclude that these *idioms* represent a way of emulating a language feature that is not natively supported (e.g., multi-methods in this case).

Cwalina and Abrams [14] find that the common design is a great advantage of the .Net library, because it makes the platform easier to learn. In addition, following and sticking to a set of conventions is beneficial, because once a developer has understood the conventions in one part of the framework, they can be projected to other areas of the platform as well. This makes programs easier to understand and also shortens the initial learning phase for new developers.

Baxter et al. [7] looked at the 'shape' of Java Software, finding that certain metrics follow power laws. They later extended their findings in [40] using a large-scale corpus of software, not unlike eariler efforts by Bajracharya et al. [5], with both these works serving as pioneering work in studying language design on a larger scale. Bloch [10] investigated effective ways to write Java programs. They say that in addition to the syntax of a program and the vocabulary available, it is crucial to note that the community shapes the actual usage of the language. Knowing about patterns and idioms can improve the quality of the resulting programs and make their creation easier for the developers.

Smit et al. [35] analyzed coding conventions and their effect on maintainability. From a preliminary survey with professional software developers, they identify several coding conventions that they find to have an effect on maintainability in open-source projects.

Allamanis et al. [3] mined patterns from source code trying to learn project-specific code conventions. They show that, indeed, some style or naming conventions are so dominant that they can be automatically extracted from source code. They call these learned patterns *Natural Coding Conventions* that are not defined as strict rules, but through the majority of developers that apply them. They validate the automatically mined patterns by successfully suggesting patches that improve the consistency of a project.

In conclusion, existing research on the terms of language design, patterns and idioms yield different interpretations and opinions: no rigid understanding exists. We will use the term 'idiom' to encompass any reusable abstraction, even those which may be considered inherent language features in different contexts. We believe there is no harm in using a broad definition at this stage, because it is always possible to omit and disregard certain idioms in future research.

```python
# numbers from 1 to 999
xs = range(1, 1000)

# Non-pythonic
res = []
for index in range(len(xs)):
    if xs[index] % 2 == 0:
        res.append(xs[index] * 3)

# Pythonic
res = [x * 3 for x in xs if x % 2 == 0]
```

Figure 1. Comparison of different programming styles given a simple example: create a list containing the tripled values of all even numbers between 1 and 999.

2.2 Pythonic Code

While existing research into Python programming exists (e.g., Holkner and Harland [21] looking into dynamic behavior), to our knowledge, 'Pythonic' idioms have not been subject to existing research. Although Python comes with an official style guide (PEP 8 [42]), this style guide only references the term 'Pythonic' once, and in a very limited scope. Furthermore, introductory text books and tutorials teaching Python hardly mention the term 'Pythonic', and it appears as if learning how to write pythonic code is a skill that can be acquired only with experience. Consequently, how to write idiomatic Python code can be non-obvious to Python beginners. Listing 1 shows an example comparing a solution that may be found in other languages to the more appropriate, pythonic solution. One existing definition of the term can be found in the official Python glossary[4]:

> *An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a for statement. Many other languages do not have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead, as opposed to the cleaner, pythonic method.*

This definition indicates a broad meaning, referring to both concrete code, but also 'ideas' in a general sense. While we consider an 'idiom' to be any reusable abstractions, in this

[4] https://docs.python.org/3/glossary.html#term-pythonic

paper we define 'pythonic' even more broadly. Community members readily use the term to describe all sorts of patterns. For example, we refer to using the `finally` keyword (for blocks always to be executed after a `try`/`except` block) as being pythonic. Although `finally` is a concept present in other languages, novice Python programmers may use simpler means, like a cleanup function, in place of `finally`. Hence we can state that `finally` is *more pythonic* than other solutions to the same problem.

Despite the strong sense among Python developers that writing pythonic code is desirable, there is no empirical evidence to corroborate this feeling. In order to give actionable advice with actual positive consequences, be it when educating or guiding new developers [36, 37], refactoring existing code under review or when discussing future features [4], empirical – rather than anecdotal – evidence is required. Hence, there is a general need to investigate the community and the concept of pythonic code closer to better understand the 'Pythonic' phenomenon.

3 Research Method

To answer our research questions, we follow a mixed method approach [13], which is widely used in software engineering research [16, 33]. We conduct interviews with Python developers and combine this qualitative research with a quantitative analysis, in which we search for use of specific pythonic idioms in Python software repositories. We provide a comprehensive replication package containing the interview questions and answers as well as the empirical data and all scripts used to process and analyze it[5].

3.1 Interviews with Python Developers

For the interviews, we identify those topics of interest that cannot be (easily) answered empirically. For example, the interviews ask developers about their opinion on pythonic idioms, what they think are the most important idioms, how they learn and discuss them with other developers, and how important these idioms are in their professional life.

To find participants for our interview study, we contacted alumni of an afiliated university by email and approached developers at a well-known Python conference. We tried to compile a group of candidates that offers different perspectives on how Python developers perceive and use pythonic source code. Eventually, we performed 13 interviews with developers that have a diverse background (see Table 1) and experience with Python ranging from 1 year to more than 15 years. Interviewees work in different companies and universities, and some of them publish their code on open-source platforms like GitHub, while others are only working on closed source code.

We conducted most of the interviews remotely in a video chat, but some developers were also interviewed in person.

[5] http://tiny.uzh.ch/QF

Table 1. Characteristics of the interviewed developers. The second column gives the experience with the Python language (in years), while the third column offers an overview of their current professional activity.

Id	Python exp. (years)	Current employment
I1	6	DevOps Eng.
I2	16	Softw. Consultant, Python Trainer
I3	4	Chief Data Scientist
I4	3	SecDevOps Backend Eng.
I5	11	Researcher
I6	>6	Director of Eng.
I7	6	Software Developer
I8	2	Software Developer
I9	>10	CTO
I10	2-3	Student
I11	3	Chief Data Scientist
I12	1	Software Developer
I13	9	Infrastructure Automation Eng.

We followed an open-ended interview style [39] and each interview lasted about 15 minutes on average. At the beginning of each interview, we have always put the interview in context and introduced its purpose to the interviewees, by explaining the research we are doing on pythonic ideas, including possible future research. We had a minimal set of questions to guide the interviews (the questions are part of the replication package), but the order in which questions were asked could differ between interviewees. For example, if an answer to an upcoming question was already given before, we did not ask the question again. Our questions can be grouped into three categories: (1) those related to the meaning and concept of the term 'Pythonic', (2) how developers became aware of pythonic idioms and how they approached them, and (3) those on the impact and spread of 'Pythonic' in the Python community, including how it affects the professional environment (e.g., job opportunities and practices). All the interviews were recorded, transcribed into text, and translated into English where required. Then, we merged all related questions of each candidate to analyze them together. We also grouped the participants on different criteria (years of experience, open source-private companies, kind of companies, ...) in order to see if similarities and differences can be found in their answers.

3.2 Identification of Pythonic Idioms

For the identification of the idioms, we followed a systematic procedure, taking into consideration a variety of sources. First, we collected idioms from presentations given by renowned Python developers that frequently mention the word 'Pythonic', e.g., Hettinger [20] and Jeff Knupp[6]. We augmented our list with idioms from Jeff Knupp's ebook "Writing idiomatic Python 3.3" [23]. The number of idioms

[6] https://www.youtube.com/channel/UC8jQsBz_w948kSc7ehMRGmQ

identified in this way was significant. Second, we investigated two well-known web pages that list and explain Python idioms: "The Hitchhiker's Guide to Python" [32] and "The Little Book of Python Anti-Patterns"[15]. The former offers a set of good practices when writing Python code, with special emphasis on a pythonic way. The latter is really a list of anti-idioms that developers should avoid, although based on these, good practices (i.e., idioms) can be extracted as well. Third, we selected popular Python books, such as "Pro Python" [1], "Fluent Python" [30], "Expert Python Programming" by Ziade [43] and by Jaworski [22], "Programming Python: Powerful Object-Oriented Programming" [24], "Head First Python" [6], "Introduction to computing and programming in Python" [19], "Dive into Python 3" [29] and scanned them for terms such as *idiom*, *Pythonic*, *programming pattern*, *coding style* or *clean[er]*. We did not find many new idioms here, but could confirm several idioms that we already had in our set. We noticed that especially introductory books tend to provide an introduction to programming (as a global concept), rather than to the Python language (and its idioms) itself. Finally, we searched for books with Python *recipes*, i.e., books that offer *effective* solutions to specific programming situations. We found "Effective Python: 59 Specific Ways to Write Better Python" [34], "Python Cookbook" [25], and "Python Cookbook: Recipes for Mastering Python 3" [8]. Identifying idioms in these books was not easy, as these books are solution-focused and not didactic, i.e., they explain how to solve a more or less complex problem, but do not offer further insight into specifics of the language. However, in addition to confirming many of our idioms from other sources, we identified additional idioms (`@classmethod` and `@staticmethod`, specifically). Several interviewees stressed that using the `itertools` module is important for writing pythonic code, so we decided to include it as an idiomatically important element as well.

3.3 Measuring the Prevalence of Idioms in Open-source Code

To verify that the concrete pythonic idioms we selected for our catalog are actually used in real-world projects, we performed an empirical st looking for pythonic idioms in 1,000 Python repositories hosted on GitHub. We obtained a list of projects through the GitHub API by querying for projects where the majority of code is written in Python, sorted by the number of 'stars'. In our query, we filtered projects that were forks, archived, private, or smaller than 1mb, thus avoiding very small repositories. Furthermore, we filtered projects whose description or 'readme' contained the word 'book', and whose name contained the terms 'tutorial', 'awesome' (a common keyword to designate lists of links), 'cookbook' and 'manual'. We also did a manual ensured that no books were present in the resulting selection, in this way it is possible that we removed some projects which would have been interesting, but we wanted to make certain that we avoid repositories not containing actual coding projects. The GitHub API returns at most 1,000 projects for a given query, however they are paginated across 10 pages and multiple calls to the same page returns different results. As such, retrieving these 10 pages only once usually provides fewer than 1,000 projects because the same project may appear on different pages. For this reason, we kept querying across all 10 pages until we had at least 1,000 projects.

All idioms in our catalogue can be detected by analyzing ASTs. Some idioms, like list comprehensions, `with` statements or Lambdas can be detected directly from the presence of the corresponding AST node types used by the parser, without any risk of false positives. Others, like 'enumerate' or 'OrderedDict' are function calls, thus the name of the called function is matched. Magic methods are naturally detected by looking for function definition nodes with the appropriate name. To make these detections, we utilized LISA, a framework for performing large-scale software analysis on abstract syntax trees[2]. We analyzed the most recent revision of all 1,000 projects, totalling 178,735 files containing 38,505,577 lines of Python code. For each project, we recorded the number of occurrences of each idiom. Then, we aggregated them to determine the number of projects each individual idiom occurs in at least once, as well as the total number of occurrences in all projects.

4 Results

In this section, we discuss the results of our studies and discuss the first two research questions. The interviews, summarized and interpreted in Section 4.1, provide answers to RQ_1, i.e., how developers perceive, learn, and value the term 'Pythonic'. In section Section 4.2 we provide a catalog of idioms we identified and subsequently found in open-source projects to answer RQ_2.

4.1 Interviews

We structure the results according to the three different topic outlined in Section 3.1, namely the meaning of 'Pythonic', how it is learned, and what its impact is on the community and at work.

4.1.1 On the Meaning and Concept of 'Pythonic'

All 13 interviewees knew about the term 'Pythonic'. Respondents understood 'Pythonic' to be *elegant* and *readable* code that makes use of constructions that are provided either by the language or by its standard library. When they talked about 'Pythonic', they pointed out that it boosts readability and performance. Many of them argued that coding the pythonic way was the most *accepted* way to code by the Python community:

> *[Pythonic code] is code that adheres on the one hand to the principles of the Python Zen, which can be seen when doing "import this". [I1]*

The most frequent pythonic example given were List Comprehensions (10 times). The rest mentioned using the `in` keyword, rules from the Python Style Guide [42], using `for`s and some other built-in functions like `enumerate`. Depending on the experience with Python, interviewees explained 'Pythonic' in slightly different ways. Those who are more experienced mentioned built-ins, efficiency, and structure more frequently, while the less experienced described it as a way of obtaining better styled code and requiring a lower number of lines of code for solving a problem.

Regarding other languages, interviewees commented that idioms can also be found in languages like GOLANG. However, even though idioms exist in other programming languages, there is not always one typical way to code a task as there is supposed to be in Python:

> *In Python maybe there is a greater tendency to value that. In other languages, it is a quality that is not pursued by the community. But [in] Python, from its origin, perhaps by chance or by its creation, I do not know, it is highly valued. It is a language that has many possibilities and more advanced functionality that makes many idioms. It is something that comes from the language, and in Python there is more [of it] than in other languages. [I2]*

Another response points out that Python is a language focused on readability, and therefore there are strongly recommended practices making the code easier to understand, usually leading to fewer lines of code:

> *'Pythonic' is more like using the right tool at the right place, and by "right tool", I mean everything that's provided by the language and its standard library. [I5]*

Some indicated, however, that idiomatic code need not be synonymous with 'Pythonic'; in their opinion, idioms are merely a means for making code more pythonic:

> *While there are many idioms in Python, using them does not mean that you're writing pythonic code. Sometimes, idioms make the code less readable, or more complicated. For instance, using* reduce *can be seen as idiomatic, but in general [it] results in code that is not easy to understand, and thus is not pythonic to my eyes. [I5]*

There is an emphasis, especially among the more experienced developers, that in Python there is an easier way to code a task with idioms and style. In comparison, other languages may have certain conventions about a consistent or an appropriate style, but not the way to solve a problem:

> *In python, the most pythonic way is the clearest, the best or even the most optimal one in a computational sense, and that makes it a little better to use. [I2]*

All interviewees told us that pythonic code was desirable, and that the most important characteristic about pythonic code is to make the code easier to understand and maintain. In their opinion, being pythonic helps to detect errors and even to make fewer mistakes.

4.1.2 On How Programmers Get to Know Pythonic Idioms

Most of the interviewees answered that they learned about pythonic code from reading code in repositories of other projects. They also got to know pythonic code from books, conferences and from colleagues at work. STACKOVERFLOW is often referred as the best way to find a pythonic idiom, although sometimes it is not the best source to learn (and understand) those idioms:

> *I have read many books [on Python and 'Pythonic']. There are times when you find yourself stuck, then StackOverflow shows you multiple points of view of people, and you always learn. [I4]*

Becoming a pythonic programmer takes time. In this regard, interviewees acknowledged that some idioms are easier to understand (and use) than others:

> *I did not understand [the pythonic idioms], but I was interested in the documentation and a world opened up for me. It took me a while to take advantage of it. [I10]*

Most of them think that understanding and reading idioms is easier than implementing them. As an example of difficult idioms, they indicated decorators and some functions from the *itertools* module (a library for efficient iteration of collections). All experienced programmers interviewed agreed that their code became more pythonic year after year. When asked if they can differentiate a beginner programmer from an advanced programmer by looking at their code, some stated that the usage of 'Pythonic' idioms is a good signal:

> *'Junior programmers are those who write simple code with some* errors*; intermediate programmers use a lot of tools from the Python library, but their code is hard to understand; mastery is achieved by those programmers who program simple code, readable and use a lot of idioms. [I10]*

Python beginners can sometimes be identified from the use of *camelCase* instead of *snake_case* in variable or function names as well as the implementation of *getters* and *setters*, indicating that these developers come from another language. Among Python developers, those who use more pythonic idioms are seen as having more expertise in the Python language:

> *'Pythonic' and Python idioms are different things. [However,] 'Pythonic' can be used to measure a developer's skills, idioms can be used to (at least) measure a developer's knowledge. [I5]*

4.1.3 On the Impact and Propagation of 'Pythonic' in the Python Community and at Work

In companies creating open-source software, it is usually mandatory to develop pythonic code (either directly or because of peer review). In companies where the code is not open source, interviewees stated that code is supposed to be documented, but not necessarily pythonic. When we asked whether pythonic code is positively viewed in a work interview, most of the interviewees agreed that writing in a

pythonic way is usually associated with high expertise in the language:

> *Yes, because it gives an idea of what you know about Python [I12]*

An interviewee argued that if you are not a good Python programmer, it is better to show your general programming skills, and to focus on the goal of the exercise instead of trying to write pythonic code, as this could be counterproductive.

The interviewees generally agree that they try to make source code more pythonic when they see something that is hard to understand. Peer review is common in open-source projects and in this process pythonic idioms are frequently introduced. But once integrated into the repository, code tends not to be changed without reason. The majority of the interviewees admit that they usually do not go back to adapt old code to a newly discovered new idiom:

> *[When I learn a new idiom], I'm not looking at all the previous lines [I have written]. I incorporate it into my toolbox and then when I touch something, I modify it and leave it better, [...] but it's not an obsession. [I2]*

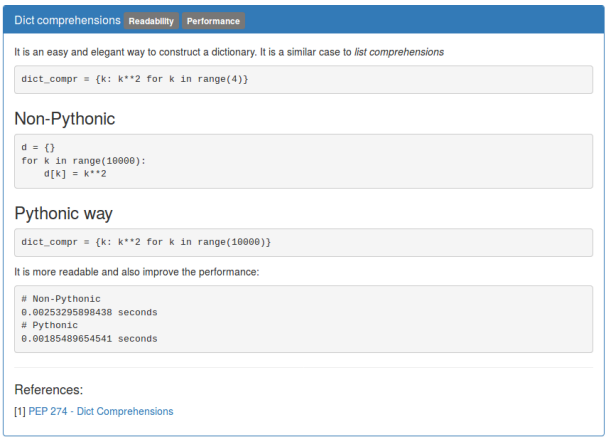

Figure 2. Dict comprehension; one of the entries in the online catalogue of idioms.

4.2 Pythonic Idioms

The list of idioms we obtained from our research is shown in Table 2. We have classified the idioms following two characteristics: readability and performance. There are idioms that have been conceived to make Python code more readable, by abstraction, shortening or added syntactic *sugar*. Idioms can also be more efficient than a more basic solution and some idioms are both more readable and more efficient. To present the idioms in an accessible manner, we have created an online catalog with all the collected idioms[7]. For each idiom, we provide a name, an explanation, an example, whether is is more readable or more efficient (sometimes including a comparison), and references with further information. Figure 2 shows the entry for dict comprehensions, an idiom that has been classified as being both more readable and more efficient than its non-recommended anti-pattern.

Table 2. Non-exhaustive list of pythonic idioms. The tags, P and R, indicate whether the idiom improves performance and/or readability. Given our sample of 1000 popular Python projects, the last two columns indicate how many of them incorporate the idiom, as well as the total count in all projects.

Idiom	# projects	total
List comprehension P R	866	75,466
A concise way to create new lists from other sequences, e.g., applying an operation to each element (like 'map' in other languages), or selecting elements that satisfy a condition (like 'filter' in other languages).		
Dict comprehension P R	146	796
The counterpart to list comprehensions but for creating dictionaries (a.k.a. 'Maps' in other languages).		
Generator expression P R	709	33,038
Like list comprehensions, but for creating sequences lazily (elements are allocated when accessed).		
Decorator R	765	114,545
Python provides facilities for the 'Decorator' design pattern. Decorators wrap existing classes or functions to alter their functionality dynamically.		
Simple magic methods	759	78,376
Intermediate magic methods	417	13,255
Advanced magic methods	190	2,613
Functions like `__str__` (with a name surrounded by double underscores) are called 'magic'. For an example, see the 'total_ordering' idiom at the bottom of this table. We group these methods into three levels of advancedness based on the interviews and their commonness in source code. We do not count `__init__` (used to define class constructors) because it is extremely common.		
finally R	504	18,900
An optional `finally` block succeeding a `try`/`catch` clause is always executed.		
with R	848	143,453
The with statement starts a code block providing a handle (typically for a file) which is automatically closed at the end of the block. For example, `with open('/tmp/file') as f: ...` replaces `f = open('/tmp/file'); ... ; f.close()`.		
enumerate R	681	19,453
Calling enumerate on a sequence returns an interable where each element is a tuple containing an index and the original element.		

[7] http://pythonic.libresoft.info/idioms

yield P 672 56,687
The `yield` keyword is used to implement generators (i.e., lazily evaluated data structures) in place of using `return`. It provides an easier way of writing generators compared to implementing an iterator (using the `__next__` magic method).

lambda P 664 109,600
The `lambda` keyword constructs anonymous functions.

collections.defaultdict P R 314 2,930
A defaultdict is like a regular Python dictionary, but when accessing a non-existent key, a default value is added and returned instead of raising an exception.

collections.namedtuple R 262 2,211
A factory function for creating tuples where individual fields are addressable not only by index but also by an attribute name.

collections.deque P R 180 1,698
The deque (double-ended-queue) is a generalization of queues and stacks in Python.

collections.Counter R 133 1,074
Counter is a subclass of Python dictionaries providing additional functionality for counting elements based on certain attributes.

@classmethod R 518 22,220
In Python, the first argument to any regular function implemented in a class receives the *instance* as the first argument (usually called 'self'). A function decorated with `@classmethod` receives the *class* as the first argument instead.

@staticmethod R 487 11,552
In contrast to regular class functions, which always receive the instance as the first argument ('self'), functions decorated with `@staticmethod` do not receive any default arguments.

zip P R 554 14,929
`zip` takes two or more iterables and returns a new iterator where each i'th element yielded is a tuple containing the i'th elements of each input iterable until at least one input is exhausted.

itertools P R 128 839
The itertools module contains several functions (e.g., `zip_longest`, `starmap`, `tee` and `groupby`) which provide common functionality concerning iterables, which a novice would likely implement by hand.

functools.total_ordering, R 30 82
A class decorated with `@total_ordering` must implement the `__eq__` (equals) magic method and at least one of the `__lt__` (<), `__gt__` (>), `__le__` (≤), or `__ge__` (≥) comparators. The decorator automatically infers the remaining functions, allowing instances of this class to be ordered accordingly.

4.3 Threats to Validity

The work presented in this paper was carefully planned and executed, but several threats to validity exist for our results. In the following, we will discuss them and our mitigation strategies.

Threats to the *construct validity* concern the way in which we set up our study. To mitigate potential issues we performed direct interviews with developers instead of a sending a survey to a larger population. On a positive side, this choice reduces the risk of receiving imprecise or unclear answers, since questions in surveys can be answered partially or superficially. On the other hand, insights based on the interviews of 13 developers may not be representative of a larger population. However, the diversity of the participants involved and the consistency of the collected answers give us confidence regarding the validity of our conclusions. The catalog of idioms we created is based on a multitude of sources but may be incomplete. Since there is no clear definition of pythonic idioms, some concepts we included may not be considered such under certain assumptions. Our detection code may be defective, although we wrote exemplary code containing all idioms and ensured that our implementation detects and counts them correctly. When detecting certain idioms which are counted based on the name of a function call, there is a small risk that some developers may sporadically go against convention and implement their own function with the same name.

Threats to the *internal validity* concern confounding factors that could influence our results. Participants might be influenced by the way we structured our interviews and the type of questions we asked them. Thus, we primarily used failry general, open questions. This reduces the potential bias imprinted upon the subjects by the interviewer, since they have a chance to answer without premeditated hints on what answers the interviewer may expect [33].

Threats to the *external validity* concern the generalizability of our results. The choice of interviewees in our study may contain bias and thus not represent the global software landscape at large. All our participants were male and working in Spain, and although we do not suspect significant regional differences, they cannot be ruled out. Furthermore, a sample size of 13 participants is not very large, though not particularly low compared to other studies in software engineering (e.g., [26, 44]). Given that we recruited both experienced and novice developers from several sources (universities and an international Python conference), we are confident to have mitigated this issue as far as possible. It remains an open question how translatable our findings are to other communities and future work is certainly necessary to solidify our findings.

5 Discussion

In this paper, we lay the groundwork for exploring and understanding what it means to have a dedicated term describing quality of craftsmanship within a programming ecosystem. What is meant when people say the word: 'Pythonic'? Does it refer to idiomatic code? Design patterns? A common understanding of problems and solutions? Beyond this, we wonder

about the broader implications summarized in RQ_3. How does it affect communication within the community? And could the idea of a language-specific quality 'brand' be translated to other languages or ecosystems? In this section we discuss our findings and provide a vision of how *anchoring* a common understanding of principles and behavior in a community can be accomplished by means of such a dedicated term.

From our interviews, we learned that there is ample awareness of the importance of writing pythonic code among Python developers, even though the term is understood in slightly different ways from person to person. Developers, especially experienced ones, point out that 'Pythonic' goes beyond the use of idioms, which are just a means to a more important goal: code that is easy to read and maintain. Interviewees agree that it takes time to master pythonic code. Pythonic mastery is highly related with being socially active in the Python community: developers learn pythonic ideas from reading other's code or during the peer review process. Informal support, such as the one found on StackOverflow, is highly valued. Finally, developers report that creating pythonic code offers evidence of Python mastery, which can be beneficial from a professional point of view.

The term 'Pythonic' is used liberally in the Python community, but a thorough explanation of its meaning is hard to find and defining it is difficult. It certainly exhibits multiple facets: for one thing, people use it to refer to specific structures in code, like those we offer in our catalog of idioms. We can say that non-pythonic code "looks wrong" to an experienced Python developer for whom the pythonic version is more readable. However, we also saw discussions of the 'Pythonic-ness' of large-scale structures and concepts, such as the design of libraries or frameworks. This shows that the scope of the term 'Pythonic' appears to go far beyond concrete source code and that it also refers to a way of thinking about problems and potential solutions. Due to its lacking of a clear definition, it is not possible to make a statement on whether or not the term's meaning has changed with time.

Given the loose definition of the term 'Pythonic' despite its widespread use, we surmise that it refers to a common underlying *culture*. Paige et al. understand culture as "the shared patterns of behaviors and interactions, cognitive constructs and understanding that are learned by socialization." and say that "it can be seen as the growth of a group identity fostered by social patterns unique to the group" [27]. In the Python community, the awareness of this culture seems to be particularly strong: *the community even has a specific, widely used term for it!* People strive to "think Pythonic" and create software that fits the culture from the get-go.

The consequences of a shared culture, concretized by a poignant term like 'Pythonic', are significant: culture offers a signaling effect, i.e., those who have more culture are viewed by others as more competent and thus have higher chances of being involved in decisions and future prospects [31]. As with natural language, we can identify different "registers", but only cultivated people are able to understand and express themselves in all of them [9], from beginner to pythonic code. The word 'Pythonic', ever-present in almost any discussion on how to apply the programming language, serves as a constant reminder that source code should be readable, maintainable and of high quality in general.

The insights we gained by observing the Python community open up several paths for future research, both empirically, on the level of source code and implementation, as well as in the cultural, social sphere.

On the *source-code level*, future work can be devoted to investigating the effect and practical use of idiomatic source code. By analyzing the evolution of source code in open-source repositories we can learn how the *usage of different idioms changes over time* and *how new idioms are introduced and shared* within the community. In addition, empirical approaches can be explored to identify developers who have achieved a certain cultural level in software projects, offering us the possibility of discovering *learning paths* for beginners from paths previously taken by more experienced developers. Another interesting line of research is related to the investigation of cultural effects of apply pythonic concepts on other program languages or software communities and whether 'pythonic code' is actually easier to comprehend and maintain. It would be valuable to know the effect of idiomatic source-code on program comprehension or on the 'bugginess' of source code. If a positive effect can be verified, the creation of tools that help developers to learn and migrate to more idiomatic source-code is a worthwhile research direction.

On the *cultural level*, a natural next step is to contrast practices from the Python community with other communities, both those that do have a guiding motto, and those which do not. For example, it can be argued that Google tried to introduce, with some success, a term like 'Pythonic' for its Android UI design, namely 'material design'[8]. The term is now used in discussions on Android UI development as a broad principle, not just a set of guidelines (indeed Google says that 'Material is the metaphor'). The Archlinux community uses 'Keep It Simple, Stupid' (KISS) as their cultural mantra [11]. Having a shared motto can also transport values, which can support decision processes, like Facebook's famous motto 'Move fast and break things'. Relating how these terms evolved in different ecosystems could teach us how to effectively introduce and curate such terms more actively in communities lacking them. Furthermore, it would be worthwhile to learn whether languages or ecosystems develop more effectively or whether they spread more quickly, if the community is involved at a cultural level. Finally, it

[8] https://material.io/design/introduction/

would be useful to know the effect the term has on individual developers: is project on-boarding easier? Do developers stay in projects or ecosystems longer? Do newcomers encounter fewer socio-technical barriers when joining a software project? [36, 37].

6 Conclusion

This paper works towards understanding the meaning and effect of explicitly naming the cultural foundation of a programming ecosystem. Specifically, we point to a sociotechnical phenomenon of the Python community in which a culture has been orchestrated around the programming language under a series of loosely-defined principles and values, captured under the umbrella of the term 'Pythonic'. These principles are learned and constructed socially and not only expressed by the use of widely accepted constructs (i.e., idioms) in code, but also used as a guiding principle in any programming-related decision. We have verified, in addition, that this phenomenon seems to be fed back as in a network effect [38, 41], becoming increasingly relevant to the point of being considered a differentiating factor for the expertise of a Python developer. Although what we have reported is currently specific to the Python community, our hypothesis is that there are many software development communities that have comparable intrinsic characteristics, especially around modern programming languages (e.g., Scala, Ruby), and have similar external conditioning, e.g., tools, such as StackOverflow or GitHub, and methods, such as modern code review and pull requests. This makes us believe that many of the issues raised could be observed and investigated in environments other than the Python ecosystem.

We show that a powerful term like 'Pythonic', despite – or even because – it is lacking a specific definition, positively influences the way developers write and talk about code. We think that actively curating such language may be a contributing factor for the popularity and success of a programming language and believe that both researchers and practitioners can profit from deeper insights into this phenomenon in the future.

Acknowledgments

We thank the reviewers for their valuable feedback. This research is partially supported by the Swiss National Science Foundation (Projects №149450 – "Whiteboard" and №166275 – "SURF-MobileAppsData") and the Swiss Group for Original and Outside-the-box Software Engineering (CHOOSE).

The research of S. Panichella is also funded by Innosuisse (Swiss Innovation Agency, project MOSAIC/19333.1). J. J. Merchante's research is possible thanks to the support of the Consejería de Educación, Juventud y Deporte de la Comunidad de Madrid and the European Social Fund.

References

[1] Marty Alchin. 2010. *Pro Python.* Apress.
[2] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. 2018. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering* (05 Jul 2018). https://doi.org/10.1007/s10664-018-9630-9
[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *International Symposium on Foundations of Software Engineering*.
[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering, ICSE 2013*. 712–721.
[5] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 681–682. https://doi.org/10.1145/1176617.1176671
[6] Paul Barry. 2016. *Head First Python: A Brain-Friendly Guide.* " O'Reilly Media, Inc.".
[7] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. 2006. Understanding the Shape of Java Software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 397–412. https://doi.org/10.1145/1167473.1167507
[8] David Beazley and Brian K Jones. 2013. *Python Cookbook: Recipes for Mastering Python 3.* " O'Reilly Media, Inc.".
[9] Basil Bernstein. 1960. Language and social class. *The British journal of sociology* 11, 3 (1960), 271–276.
[10] Joshua Bloch. 2008. *Effective Java (The Java Series).* Prentice Hall.
[11] Jose Dieguez Castro. 2016. Arch linux. In *Introducing Linux Distros*. Springer, 235–252.
[12] James William Cooper. 2000. *Java design patterns: a tutorial.* Addison-Wesley Professional.
[13] John W Creswell and J David Creswell. 2017. *Research design: Qualitative, quantitative, and mixed methods approaches.* Sage publications.
[14] Krzysztof Cwalina and Brad Abrams. 2008. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries.* Pearson.
[15] Andreas Dewes and Christoph Neumann. 2018. The Little Book of Python Anti-Patterns. https://goo.gl/xQQNE2. Accessed: 2018-04-21.
[16] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*. Springer, 285–311.
[17] Joseph Gil and David H Lorenz. 1997. Design Patterns vs. Language Design. In *European Conference on Object-Oriented Programming*.
[18] Mark Grand. 2003. *Patterns in Java: a catalog of reusable design patterns illustrated with UML.* John Wiley & Sons.
[19] Mark J Guzdial and Barbara Ericson. 2015. *Introduction to computing and programming in Python.* Pearson.
[20] Raymond Hettinger. 2013. Transforming Code into Beautiful, Idiomatic Python. https://goo.gl/wgeAvp. Accessed: 2018-04-21.
[21] Alex Holkner and James Harland. 2009. Evaluating the Dynamic Behaviour of Python Applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91 (ACSC '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 19–28. http://dl.acm.org/citation.cfm?id=1862659.1862665
[22] Michal Jaworski and Tarek Ziadé. 2016. *Expert Python Programming.* Packt Publishing Ltd.
[23] Jeff Knupp. 2013. *Writing Idiomatic Python 3.3.* CreateSpace.
[24] Mark Lutz. 2010. *Programming Python: Powerful Object-Oriented Programming.* " O'Reilly Media, Inc.".

[25] Alex Martelli, Anna Ravenscroft, and David Ascher. 2005. *Python cookbook*. " O'Reilly Media, Inc.".

[26] S. McKee, N. Nelson, A. Sarma, and D. Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 467–478. https://doi.org/10.1109/ICSME.2017.53

[27] R Michael Paige, Helen L Jorstad, Laura Siaya, Francine Klein, Jeanette Colby, D Lange, and R Paige. 2003. Culture learning in language education. *Culture as the core: Perspectives on culture in second language learning* (2003), 173–236.

[28] Alan J Perlis and Spencer Rugaber. 1979. Programming With Idioms In APL. In *ACM SIGAPL APL Quote Quad*.

[29] Mark Pilgrim and Simon Willison. 2009. *Dive Into Python 3*. Vol. 2. Springer.

[30] Luciano Ramalho. 2015. *Fluent Python: clear, concise, and effective programming*. " O'Reilly Media, Inc.".

[31] Mari Rege. 2008. Why do people care about social status? *Journal of Economic Behavior & Organization* 66, 2 (2008), 233–242.

[32] Kenneth Reitz and Tanya Schlusser. 2016. *The Hitchhiker's Guide to Python: Best Practices for Development*. "O'Reilly Media".

[33] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. *Case Study Research in Software Engineering*. Wiley-Blackwell.

[34] Brett Slatkin. 2015. *Effective Python: 59 Specific Ways to Write Better Python*. Pearson Education.

[35] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. 2011. Code Convention Adherence in Evolving Software. In *International Conference on Software Maintenance*.

[36] Igor Steinmacher, Tayana Uchôa Conte, Christoph Treude, and Marco Aurélio Gerosa. 2016. Overcoming open source project entry barriers with a portal for newcomers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. 273–284.

[37] Igor Steinmacher, Marco Aurélio Graciotto Silva, Marco Aurélio Gerosa, and David F. Redmiles. 2015. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information & Software Technology* 59 (2015), 67–85.

[38] Chandrasekar Subramaniam, Ravi Sen, and Matthew L Nelson. 2009. Determinants of open source software project success: A longitudinal study. *Decision Support Systems* 46, 2 (2009), 576–585.

[39] Winston M Tellis. 1997. Application of a case study methodology. *The qualitative report* 3, 3 (1997), 1–19.

[40] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference (APSEC '10)*. IEEE Computer Society, Washington, DC, USA, 336–345. https://doi.org/10.1109/APSEC.2010.46

[41] Brian Uzzi. 1996. The sources and consequences of embeddedness for the economic performance of organizations: The network effect. *American sociological review* (1996), 674–698.

[42] Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2001. *PEP 8: style guide for Python code*. Python.org. https://goo.gl/crVen9.

[43] Tarek Ziadé. 2008. *Expert Python Programming*. Packt Publishing Ltd.

[44] Manuela Züger, Sebastian C. Müller, André N. Meyer, and Thomas Fritz. 2018. Sensing Interruptibility in the Office: A Field Study on the Use of Biometric and Computer Interaction Sensors. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 591, 14 pages. https://doi.org/10.1145/3173574.3174165