Pythonic vs Refactorable Pythonic: On the Relationship between Pythonic Idioms and Code Quality in Machine Learning Projects

Gerardo Festa,[1] Giammaria Giordano,[1] Valeria Pontillo,[2]
Massimiliano Di Penta,[3] Damian A. Tamburri[3,4], Fabio Palomba[1]

[1] *Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy* — [2] *Gran Sasso Science Institute (GSSI), L'Aquila, Italy* — [3] *University of Sannio, Italy* — [4] *JADS/NXP Semiconductors, Netherlands*
g.festa22@studenti.unisa.it, giagiordano@unisa.it, valeria.pontillo@gssi.it, dipenta@unisannio.it, datamburri@unisannio.it, fpalomba@unisa.it

Corresponding author: Giammaria Giordano

Pythonic vs Refactorable Pythonic: On the Relationship between Pythonic Idioms and Code Quality in Machine Learning Projects

Gerardo Festa,[1] Giammaria Giordano,[1] Valeria Pontillo,[2]
Massimiliano Di Penta,[3] Damian A. Tamburri[3,4], Fabio Palomba[1]

[1] *Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy* — [2] *Gran Sasso Science Institute (GSSI), L'Aquila, Italy* — [3] *University of Sannio, Italy* — [4] *JADS/NXP Semiconductors, Netherlands*
g.festa22@studenti.unisa.it, giagiordano@unisa.it, valeria.pontillo@gssi.it, dipenta@unisannio.it, datamburri@unisannio.it, fpalomba@unisa.it

Abstract

Context: Python is increasingly becoming the *lingua franca* for developing Machine Learning (ML) systems, thanks to a rich ecosystem of libraries and an emphasis on readability. In this context, *Pythonic* idioms are seen as stylistic conventions that support maintainable and efficient code. Conversely, *Refactorable-Pythonic* idioms refer to patterns that can be refactored into more idiomatic Python, improving code quality in terms of maintainability, performance, and clarity.

Objective: While the assumptions about idiomaticity are widely accepted in practice, the extent to which *Pythonic* or *Refactorable-Pythonic* idioms relate to software quality in ML projects has not been systematically validated. To address this lack of empirical evidence, this paper conducts a large-scale study to assess how *Pythonic* and *Refactorable-Pythonic* idioms are related to code quality in ML systems.

Method: We analyze 303 open-source Python projects from the NICHE dataset, distinguishing between "well-engineered" (i.e., projects that adopt structured development practices such as testing, CI, documentation, and packaging) and "non-engineered" (i.e., projects that lack such characteristics). Our analysis proceeds in two main phases: (i) idiom detection, where we extract *Pythonic* and *Refactorable-Pythonic* code patterns using a combination of existing and custom detectors; and (ii) quality assessment, where we detect Python-specific smells and relate them to code metrics and other quality indicators.

Result: Truth Value Test and *Assign Multiple Targets* are the most common *Pythonic* and *Refactorable-Pythonic* idioms, respectively. In "well-engineered" projects, both idiom types positively correlate with Python-specific code smells, suggesting that idiomatic usage does not always align with higher code quality. In contrast, in "non-engineered" projects, the presence of smells is more strongly influenced by structural factors such as the number of lines of code, complexity, and commit activity.

Conclusion: We conclude by distilling lessons learned, implications, and future research directions.

Keywords: Software Quality, Software Engineering for Artificial Intelligence, Software Maintenance and Evolution, Empirical Software Engineering.

1. Introduction

As Machine Learning (ML) becomes increasingly widespread, so too does the textitasis on producing code that is readable, maintainable, and scalable [1]. Python—currently the dominant language in ML development—encourages using *Pythonic* idioms: coding practices that leverage the language's expressive syntax to promote simplicity, clarity, and efficiency [2]. For example, a typical *Pythonic* idiom, the list comprehension, replaces a manual loop for computing the squares of a list of numbers with the more concise function `[x**2 for x in range(10)]`. These idioms are widely promoted within the Python community and development guidelines as a means to write cleaner, more maintainable code [3, 4].

However, while *Pythonic* idioms are commonly recommended in practice, their relationship with software quality has not been empirically validated. Some idioms, although concise, may inadvertently foster patterns that lead to maintainability issues. Consider, for instance, the *Assign Multiple Targets* idiom, where multiple variables are assigned in a single line (e.g., `a = b = 0`). While this syntax is compact and valid, the idiom may inadvertently promote a coding style in which shared mutable state is accessed across extended blocks of code, making it more challenging to isolate responsibilities and encapsulate logic. Moreover, the idiom can encourage the propagation of multiple interdependent variables into downstream functions, increasing the complexity of function interfaces and making the code harder to maintain and evolve.

These risks are particularly relevant in ML systems, where functions are frequently used to configure models or compose data processing pipelines. In such settings, the need for rapid prototyping and experimentation often leads to function signatures expanding over time, especially when variables assigned together are passed through multiple layers of the codebase. These concerns are further amplified by the fact that contributors to ML systems often come from non-software engineering backgrounds and may lack formal training in principles such as code design and maintainability [5, 6]. So, these arguments raise a fundamental question:

> **Main Question**
>
> *Does the use of Pythonic idioms correlate with higher or lower code quality in real-world ML codebases?*

Answering this question has important implications for both research and practice. For practitioners, empirical evidence can validate or challenge current recommendations on idiomatic coding, helping teams make more informed decisions about code conventions and training practices in ML development environments. For researchers, such findings contribute to a better understanding of how high-level coding patterns influence maintainability and may inform the design of future tools, linters, or refactoring strategies that are sensitive to the specific needs of ML projects.

This paper addresses this question through an empirical investigation into whether the use of *Pythonic* idioms correlates with software quality in ML projects. As a comparison baseline, we consider *Refactorable-*

Pythonic idioms, namely code patterns that are syntactically correct but deviate from idiomatic best practices and can be mechanically transformed into more *Pythonic* alternatives. We adopt this category as a baseline because it naturally contrasts with idiomatic usage, representing functionally correct but stylistically suboptimal code. At the same time, these idioms can be reliably and automatically detected through publicly available tools. As for code quality, we operationalize it through the presence of *code smells*, i.e., recurring design deficiencies that hinder maintainability and may signal deeper structural issues [7]. Specifically, we consider *Pythonic* smells detected using DPy [8]. Our study analyzes 303 open-source Python repositories for ML development, drawn from the NICHE dataset [9], and classified as either "well-engineered" or "non-engineered" based on established criteria, such as the presence of tests, continuous integration, or packaging metadata. We focus on nine *Pythonic* idioms and their *Refactorable-Pythonic* counterparts, examining their statistical association with code smells and how these patterns vary across the two groups.

The results of our study indicate that *Truth Value Test* and *Assign Multiple Targets* are the most common idioms in *Pythonic* and *Refactorable-Pythonic* idioms, respectively. In addition, we find a statistical relationship between idioms and the presence of Python-specific code smells, especially in "well-engineered" projects, suggesting that the adoption of these idioms is not in all cases the best solution to improve code quality in ML systems. To summarize, our paper makes the following contributions:

1. A large-scale empirical study investigating the adoption of nine *(Refactorable-)Pythonic* idioms across 303 ML projects and how such idioms vary with codebase size and density;
2. A statistical analysis exploring the relationship between *(Refactorable-)Pythonic* idioms and Python-specific code smells, highlighting the idioms that most strongly correlate with quality issues;
3. A publicly available replication package including all scripts, datasets, and results used in our study, to ensure transparency and foster future research [10].

Structure of the paper. Section 2 summarizes the state-of-the-art and discusses the most closely related work. Section 3 details the research questions of our study and the research method applied to address them. Section 4 summarizes the results obtained. Section 5 discusses the implications of this work for the SE community. Section 6 discusses the potential threats to validity and the mitigation strategies applied and lastly, Section 7 concludes the paper.

2. Background and Related Work

This section provides the necessary information to understand the rest of the paper and summarizes the state-of-the-art in the context of code smells in Python projects.

2.1. Background and Motivation

Pythonic code refers to a style of Python programming that adheres to the idiomatic principles of the Python language, textitizing readability, simplicity, and conciseness. These principles are encapsulated in Python's unofficial mantra, *"The Zen of Python"* [2], which includes precepts such as *"Beautiful is better than ugly"* and *"Readability counts"*. Writing *Pythonic* code involves leveraging Python's built-in features and capabilities in a way that maximizes the language's expressiveness and minimizes code complexity.

Refactorable-Pythonic code, instead, refers to Python code that does not use these idiomatic principles and often resembles patterns that might be more common in other programming languages. This code can typically be refactored into a *Pythonic* style, which not only enhances readability but also aligns with Python's philosophy of simplicity and efficiency. The refactoring into *Pythonic* code often involves replacing cumbersome and verbose constructs with more streamlined and effective idioms.

For example, a common *Refactorable-Pythonic* approach might involve using a loop to filter active users whose age is greater than 18 and whose name starts with an uppercase "A", and collecting their names into a list. Listing 1 provides an example.

Listing 1: Filter active users with age over 18 whose names start with 'A', and collect their names.

```
filtered_names = []
for user in users_list:
    if user.is_active and user.age > 18 and user.name.startswith("A"):
        filtered_names.append(user.name)
```

A *Pythonic* refactor of the above code would use a *list comprehension* function, which is more concise and transparent, as shown in Listing 2.

Listing 2: Pythonic filter active users with over 18 whose names start with 'A', and collect their names.

```
filtered_names = [user.name for user in users_list if user.is_active and user.
    age > 18 and user.name.startswith("A")]
```

In the context of ML projects, writing *Pythonic* code can be especially impactful. For instance, ML workflows often involve dynamic configuration of models, where parameters may be generated programmatically or loaded from external sources. In such cases, idioms such as *Star in Function Call* can enhance both readability and flexibility. In the snippet provided in Listing 3, the unpacking operator * is used to pass a list of hyperparameters directly to the `LogisticRegression` constructor in a clean and maintainable way:

Listing 3: Star in Function Call example.

```
# Load hyperparameters from external config file
with open('config.json') as f:
```

```
    config = json.load(f)

# Unpack the dictionary into the model constructor
model = LogisticRegression(**config)
```

This *Pythonic* idiom not only makes the code more concise but also improves execution efficiency by optimizing parameter handling internally, representing an important advantage for data-intensive tasks typical in machine learning workflows.

In other terms, *Pythonic* idioms provide not only aesthetic and stylistic benefits but also practical advantages that may affect multiple properties of ML projects [11]. The Python's dominance in the ML domain, along with the potential benefits of *Pythonic* code, motivates our work: we aim to *empirically validate these benefits and provide a data-driven understanding of how Pythonic coding influences ML projects.*

Given the importance of using *Pythonic* code, recent research has focused on the automatic detection of refactorable-idiomatic constructs. One such tool is RIdiom [12], a framework specifically designed to identify code snippets that can be rewritten in a more idiomatic style. RIdiom addresses the challenge of promoting idiomatic usage by analyzing source code through its Abstract Syntax Tree (AST) representation. It systematically detects non-idiomatic code, i.e., *Refactorable-Pythonic*. As elaborated in Section 3.2, we employed this tool in the scope of our study, as it provides a reliable and systematic way to identify *Pythonic* idioms in real-world code. In our investigation, we focus on the analysis of code smells, which we adopt as a proxy metric to evaluate the potential impact of *Pythonic* code on software quality. In this context, several static analyzers have been proposed to detect code smells in Python codebases. Among the most prominent are PySmell [13] and DPy [8], both capable of detecting 11 Python-specific code smells with high precision and recall. While PySmell was the first tool designed for this purpose, it is no longer maintained and is currently unavailable [8]. In contrast, DPy is actively supported and accessible - which explains why we opted for this tool in the context of our paper.

2.2. Related Work

Fowler and Beck [7] originally defined *code smells* as indicators of sub-optimal design decisions that could exacerbate code complexity, particularly during maintenance and evolution tasks. A significant amount of research has been conducted on code smells in traditional software systems, *e.g.,* systems developed in Java, investigating their origins, persistence, and mitigation strategies [14, 15, 16, 17].

Further studies have correlated code smells to reusability mechanisms, such as inheritance and delegation, noting their benefits and drawbacks [16]. Giordano *et al.* [18] found that while certain design patterns negatively correlated with code smells, others positively correlated with their presence. Although the majority of this research has concentrated on Java systems [19, 20, 21, 22], studies by Vavrová *et al.* [23] have

started to explore the prevalence and characteristics of code smells in Python, discovering that smells like *Long Method* are statistically more prevalent in Python systems than in Java.

In the context of ML projects, Chen *et al.* [24] analyzed 106 Python ML projects by introducing PySmell, a Python code smell detection tool. Their findings reinforced previous work and highlighted that the *Long Method* smell is the most prevalent one. Van Oort *et al.* [25] investigated Python-specific code smells by analyzing 74 ML projects, finding that *Duplicate Code* is one of the most widespread smells in ML projects. Jebnoun *et al.* [26] used PySmell to explore deep learning projects, discovering that *Long Lambda Expression*, *Long Ternary Conditional Expression*, and *Complex Container Comprehension* smells are more frequent within deep learning code than in traditional software code. More recently, Giordano et al. [27] conducted an evidence-based investigation into how CI mechanisms affect the emergence of code smells in ML systems, finding that they may actually lead to a significant reduction of code quality issues.

When comparing our work to those discussed above, we differentiate ourselves by explicitly focusing on the influence of *Pythonic* coding practices on the prevalence of code smells in ML projects. Unlike previous studies that predominantly examine code smells within the context of traditional software systems or general Python applications, our research directly targets ML environments where *Pythonic* practices are hypothesized to have a distinct impact on software quality. In this respect, our study contributes to the field by systematically assessing whether adopting *Pythonic* idioms leads to a measurable improvement in code maintainability and a reduction in code smells, potentially providing recommendations for ML practitioners who heavily rely on Python for implementing complex algorithms and data processing tasks.

Also, our research contributes to advancing the current body of knowledge on *Pythonic* code and its practical impact. In this respect, Alexandru *et al.* [3] explored the adoption and benefits of *Pythonic* idioms, such as *list comprehensions* and *decorators*. Their study, informed by interviews with developers and an analysis of 1,000 Python repositories, highlights that these idioms are favored for improving code maintainability, aligning with the principles outlined in *"The Zen of Python"* [2].

Zid et al. [4] conducted a controlled experiment involving 209 developers to evaluate the understandability of *Pythonic* functional constructs such as *lambdas*, *comprehensions*, and *map/reduce/filter* functions, compared to their procedural counterparts. The study revealed that procedural code was generally found to be more readable than functional alternatives.

Zampetti *et al.* [28, 29] also studied whether *Pythonic* functional constructs are more prone to induced fixes than others. They found that, in general, changes to such constructs have higher odds of inducing fixes than other changes, and that this is more likely the case for lambdas and to some extent for comprehensions.

Sakulniwat *et al.* [30] investigated *when* and *why* developers adopt *Pythonic* idioms in open-source projects. They discovered that practitioners tend to use *Pythonic* idioms during evolutionary activities and, in a non-negligible number of cases, developers improve their source code refactoring using *Pythonic* idioms.

Phan-udom *et al.* [31] released Teddy, a tool that provides *Pythonic* idioms examples starting from

Refactorable-Pythonic idioms. The main limitation of this tool is the absence of refactoring operations; it only provides examples starting from a predefined knowledge base. Zhang *et al.* [32] introduced a tool for refactoring *non-Pythonic* code into *Pythonic* one, demonstrating its efficacy in real-world applications.

Some studies investigated the performance improvement achieved when refactoring code towards *Pythonic* idioms. Leelaprute *et al.* [33] provided evidence that *Pythonic* idioms could significantly enhance both memory usage and execution times, suggesting areas for further practical research. Zhang *et al.* [34] leveraged their refactoring tool and showed that *non-Pythonic* code refactored into *Pythonic* one led to performance improvements. However, Zid *et al.* [35] found that performance improvements are usually negligible for real-world source code elements, and become tangible only when the refactoring action is amplified.

While the work above discusses the relationship between *Pythonic* constructs and software quality, we further analyze this relationship for ML programs.

Similarly, some work investigates the use of Python in ML-intensive projects. Nagpal and Gabrani [36] investigate the suitability of Python for scientific applications, highlighting its advantages because of a simpler syntax and portability. Instead, we focus more on *Pythonic* idioms, and the extent to which they relate (or not) to smells in ML code.

3. Research Method

The *goal* of this study is to understand whether the adoption of *Pythonic* idioms in ML projects correlates with better software quality, measured through the presence of code smells. The focus on quality lies in assessing how idiomatic coding practices may influence maintainability-related decisions in ML systems.

The *perspective* is for both developers and researchers; the former are interested in understanding whether their coding practices are beneficial or detrimental to the quality of their software, while the latter aim to deepen their knowledge of both the adoption of *Pythonic* idioms in ML projects and the relationship between idioms and code smells. To guide our analysis, we formulate the following research questions.

> **RQ$_1$. On the diffusion of *(Refactorable-)Pythonic* idioms.**
>
> *To what extent do (Refactorable-)Pythonic idioms occur in ML systems?*

RQ$_1$ serves as a preliminary investigation to characterize how *Pythonic* and *Refactorable-Pythonic* idioms are used in practice. This question is instrumental for the subsequent analysis of the potential relationship between *Pythonic* idioms and code quality. In particular, understanding the frequency and context of idiom usage helps interpret any correlation with code smells and supports a more detailed view of their role in real-world systems. To explore idiom adoption in more detail, we split **RQ$_1$** into three sub-questions:

RQ$_{1.1}$ What are the most frequent *(Refactorable-)Pythonic* idioms in ML projects?

$RQ_{1.2}$ How does the adoption of *(Refactorable-)Pythonic* idioms vary as the project grows in size?

$RQ_{1.3}$ What is the density of *(Refactorable-)Pythonic* idioms in source code?

> **RQ$_2$. On the relationship between *Pythonic* idioms and code smells.**
> *What is the relationship between the usage of Pythonic and Refactorable-Pythonic idioms and the presence of code smells in ML systems?*

The goal of **RQ$_2$** is to investigate whether the use of *(Refactorable-)Pythonic* idioms is statistically correlated with the presence of code smells. Building on the descriptive insights from **RQ$_1$**, this analysis aims to determine whether certain idioms tend to co-occur with higher or lower smell incidence. This question lies at the core of our study, as it empirically tests the assumption that *Pythonic* idioms inherently lead to cleaner, higher-quality code. Considering the two categories of projects *i.e.*, "well-engineered" and "non-engineered", we decided to split the **RQ$_2$** into two sub-questions. Indeed, we asked:

$RQ_{2.1}$ What is the relationship between the usage of *Pythonic* and *Refactorable-Pythonic* idioms in "well-engineered" projects and the presence of code smells in ML systems?

$RQ_{2.2}$ What is the relationship between the usage of *Pythonic* and *Refactorable-Pythonic* idioms in "non-engineered" projects and the presence of code smells in ML systems?

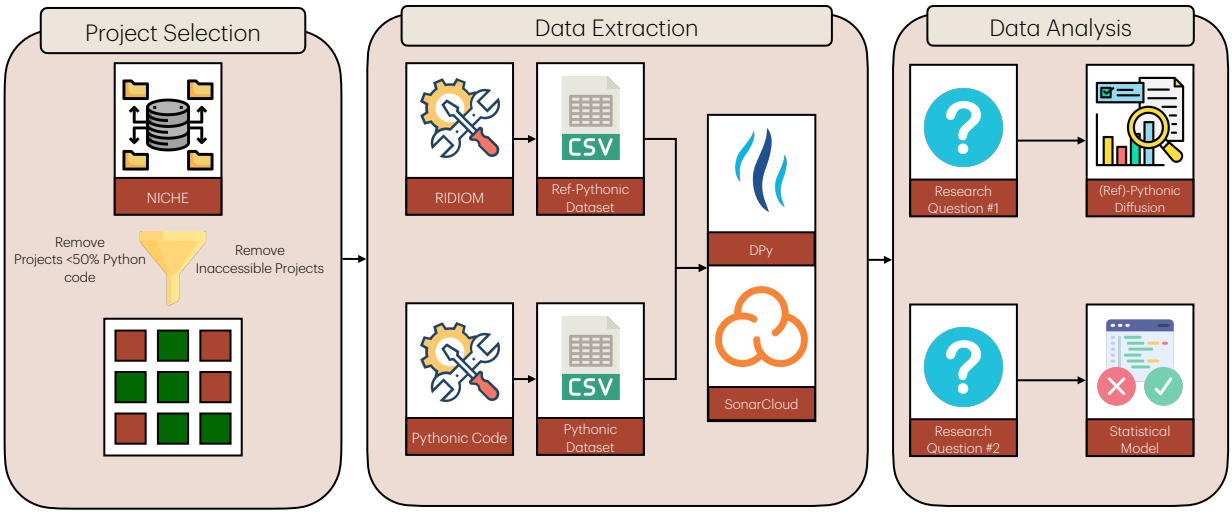

Figure 1: Research Method Overview.

Our investigation has a statistical connotation. We adopted the guidelines of Wohlin *et al.* [37] in terms of reporting and the *ACM/SIGSOFT Empirical Standards*;[1] - we leveraged the "General Standard", "Data

[1]Available at:`https://github.com/acmsigsoft/EmpiricalStandards`

Science" and "Repository Mining" guidelines. As overviewed in Figure 1, we initially mined ML projects from GitHub. Next, we simultaneously ran RIdiom to extract code snippets that could be refactored into a *Pythonic* way (*i.e., Refactorable-Pythonic* idioms) and built and tested a homemade tool to detect *Pythonic* snippets in repositories based on AST. After obtaining *Pythonic* and *Refactorable-Pythonic* idioms, to address RQ_1, we computed idioms' frequency, the variation, and density. Finally, we ran SonarCloud [38] to extract information regarding project metrics *i.e.,* lines of code (LOC), file complexity, and number of commits, DPy [8] to extract Python-specific code smells from repositories, and applied statistical tests to analyze the relationship between *(Refactorable-)Pythonic* idioms and the frequency of code smells.

Table 1: Description of the Eight Dimension Metrics that Characterize Well-Engineered Projects.

Dimension	Description
Unit testing	The project shows clearly the presence of unit tests.
Architecture	The project architecture is well-defined
Documentation	The project demonstrates sufficient documentation to enable developers to perform evolutionary and maintainability activities.
Issues	The community uses the GitHub issue management tools.
Continuous Integration (CI)	Instruments to permit CI, such as Jenkins or GitHub Actions, are regularly used in software projects.
History	The project shows a long history, demonstrating the community's maturity.
Community	The project has a large number of collaborators, showing their capabilities as a working team.
License	The project is hosted with an appropriate license, clearly defined and documented.

3.1. Dataset Description and Project Selection

The *context* of our analysis is the NICHE dataset [9], which includes 572 ML projects categorized as either "well-engineered" or "non-engineered", based on eight-dimensional metrics (see Table 1). We selected this dataset for three main reasons. First, NICHE comprises only popular, active projects with a substantial development history—specifically, projects with at least 100 GitHub stars, a minimum of 100 commits, and a most recent commit dated after May 1st, 2020. These criteria ensure the exclusion of inactive or personal repositories. Second, each project in the dataset has been manually labeled by the dataset authors as either "well-engineered" or "non-engineered", based on whether it satisfies at least four of the eight defined dimensions. This labeling facilitates a robust comparative statistical analysis between the two groups: 441 "well-engineered" and 131 "non-engineered" projects. Finally, the dataset covers diversified project domains, enhancing our findings' generalizability and relevance.

To ensure that only projects useful for our analysis are considered, we applied two additional filters *i.e.,* we removed projects with less than 50% of Python code to consider only those mainly written in Python and projects not yet accessible on GitHub (*e.g.,* projects migrated to other system versioning). As a result of these steps, we ended up with 303 projects, of which 76 were "not engineered" and 227 were

"well-engineered". Table 2 provides the descriptive statistics for the variables *Stars*, *Commits*, *NLOC*, and *Pythonic Percentage* divided according to the column *Engineered*.

Table 2: Descriptive Statistics for Variables *Stars*, *Commits*, *NLOC*, and *Python Percentage* divided according to the *Engineered* Column.

		Stars	Commits	NLOC	Python %
Well-Engineered	Min	100	102	234	52%
	Mean	1,843	1,088	20,055	91%
	Median	650	465	11,130	98%
	Max	31,057	47,094	232,930	100%
Non-Engineered	Min	111	100	396	51%
	Mean	2,033	422	15,158	89%
	Median	405	223	4,303	97%
	Max	64,439	4,914	268,628	100%

3.2. Selection of Pythonic Idioms and Code Smells

This study focuses on the relationship between *Pythonic* coding practices and the presence of code smells in ML systems. To this aim, we selected a representative set of idioms and code smells.

The *Pythonic* idioms were chosen based on their expressiveness, frequency of use, and alignment with the principles outlined in the *"The Zen of Python"* book [2]. We rely on RIdiom [12], a state-of-the-art tool for the detection of candidate idioms in Python. The tool operates on the AST and supports a catalog of idioms, including *List Comprehension*, *Dict Comprehension*, *Set Comprehension*, *Chain Comparison*, *Truth Value Test*, *Loop Else*, *Assign Multiple Targets*, *Star in Function Call*, and *For Multiple Targets*. These idioms improve code conciseness, readability, and efficiency. Table 3 presents the idioms used in this study.

As for code smells, we leverage DPy [8], a Python static analyzer capable of identifying 11 Python-specific implementation smells, such as *Long Method*, *Complex Conditional*, *Long Statement*, and *Empty Catch Block*. These smells are commonly associated with reduced readability and maintainability in codebases. As explained in Section 2, we preferred DPy over earlier tools (e.g., PySmell) due to its active maintenance and superior detection performance. Table 4 lists the smells considered in the scope of the study.

Starting from the full set of implementation smells detected by the DPy tool, we retained only those for which a mitigation strategy could plausibly involve using a *Pythonic* idiom. This filtering step ensured that our analysis focused on smells with a *direct* and *interpretable* connection to idiomatic constructs, rather than broader structural or semantic issues. The resulting mapping is presented in Table 5, organized row-wise, each row describing a single code smell instance. For each smell, the Table shows a concrete code example,

Table 3: List of *Pythonic* idioms recognized by RIdiom.

Idiom	Description
List Comprehension	A concise way to create lists in a single line by applying an expression to each item in an iterable.
Set Comprehension	A method for creating sets by directly iterating over items and applying transformations, eliminating the need for loops and 'add' operations.
Dict Comprehension	Allows building dictionaries in a single line by iterating over items and dynamically defining the keys and values.
Chain Comparison	Permits combining multiple comparison expressions in a single statement, reducing redundancy by avoiding separate if conditions.
Truth Value Test	Leverages Python's inherent truthiness rules to simplify conditions. Instead of explicitly comparing a variable to values like zero or None, it directly evaluates its truthy or falsy state.
Loop Else	Adds an 'else' clause to a loop, which runs only if the loop completes normally (i.e., without encountering a 'break').
Assign Multiple Targets	Allow assigning multiple variables in one line.
Star in Func Call	Uses the unpacking operator '*' to pass a list or tuple as multiple arguments to a function call, improving readability and flexibility when handling dynamic data.
For Multiple Targets	Enhances readability and efficiency in loops by unpacking multiple items from an iterable (e.g., tuples or lists) directly into separate variables, avoiding index-based access.

a refactored version, the *Pythonic* idioms applied during refactoring, and whether the transformation was implemented through idiomatic constructs (✅) or not (❌). Additional columns provide a brief textual description of the problem and the rationale behind the mitigation. This layout enables the reader to assess both the nature of the smells and the plausibility of using idiomatic code as a remedy. Accordingly, we included in our analysis only those smells for which the application of *Pythonic* idioms represents a plausible mitigation strategy. This choice aligns with the core objective of our study, i.e., to empirically assess whether the use of idiomatic Python correlates with measurable improvements in software quality. By focusing on smells that can be explicitly addressed through idiomatic constructs, we ensure that any observed associations (or lack thereof) can be meaningfully interpreted in terms of the role that *Pythonic* practices play in mitigating common quality issues in ML codebases.

Looking more closely at the Table, an initial hypothesis emerges: *Pythonic* idioms appear to offer natural mitigation strategies for smells that involve verbose or repetitive syntactic patterns (e.g., *Long Statement*, *Complex Method*, *Empty Catch Block*). In contrast, smells rooted in naming (e.g., *Long Identifier*,

Table 4: Code Smells Detectable Using DPy.

Implementation Smell	Description
Long statement	A line of code that is excessively long, making it harder to read and maintain.
Long parameter list	A function definition that includes too many input parameters, which can reduce clarity and increase complexity.
Long method	A method or function that performs too many tasks, resulting in low readability and poor modularity.
Long identifier	An overly verbose name for a function, class, variable, or field that impacts code readability.
Empty catch block	A catch or except block that lacks handling logic, potentially hiding errors during execution.
Complex method	A method with intricate logic or too many responsibilities, making it difficult to understand or test.
Complex conditional	A condition expression containing numerous logical operators, which can obscure intent and cause errors.
Missing default	A `match-case` statement that lacks a default case, risking unhandled inputs.
Long lambda function	A lambda expression that is too lengthy or complicated, reducing code brevity and clarity.
Long message chain	A deep chain of method calls that hinders understanding and makes debugging more difficult.
Magic number	A numeric literal used without explanation, making the code harder to interpret and maintain.

Magic Number) or structural design (e.g., *Missing Default*) do not lend themselves to idiomatic refactoring and instead require interventions beyond what idioms alone can express. This observation motivates our empirical investigation: whether the presence of *Pythonic* idioms in real-world ML codebases correlates with a reduction in detectable smells, or whether idioms serve primarily stylistic purposes with limited practical effect. In this sense, Table 5 serves not only as a reference for the kinds of issues idioms may address, but also as a conceptual bridge to our experimental design.

3.3. $\boldsymbol{RQ}_1$: On the Diffusion of (Refactorable-)Pythonic Idioms

To address $\mathbf{RQ}_1$, we proceeded from two sides. On the one hand, we needed to count the instances of snippets of code refactorable in *Pythonic* idioms. To achieve this, we employed RIdiom [12], a tool designed to identify *Refactorable-Pythonic* code by analyzing the Abstract Syntax Tree (AST). The tool was evaluated on over 7,000 Python projects and achieved 100% accuracy for all nine idioms. Table 3 describes the idioms refactorable by the tool.

To address $\mathbf{RQ}_{1.1}$, first, we instrumented RIdiom in order to count the instances of *Refactorable-Pythonic*. Then, to extract and quantify occurrences of *Pythonic* idioms in software projects, we developed a custom analysis tool. This tool takes a Python project as input, recursively processes all Python files, and constructs their corresponding ASTs. Each AST is then traversed node by node to identify code structures that match specific Pythonic idioms. For each idiom, the tool applies pattern-matching rules that resemble those used by RIdiom, and are also aligned with the principles described in the *"The Zen of Python"* book [2]. For example, to detect the *Assign Multiple Targets* idiom, the tool identifies `Assign` nodes in the AST

Table 5: List of Code Smells Detectable Using DPy, the Mitigation Strategy, and Whether a Pythonic Idiom Is Applied.

Code Smell	Example	Refactored Code	Idioms Used	Implemented?	Description	Mitigation Rationale
Long Statement	`sum([x**2 for x in range(1, 101) if x % 2 == 0 and x > 10 and x < 90])`	`sum([x**2 for x in range(1, 101) if x % 2 == 0 and 10 < x < 90])`	List Comprehension, Chain Comparison	✅	Computes the sum of squares of even numbers in a range, but the verbose bounds make the expression hard to scan.	Chain comparison shortens the numeric test, keeping the list comprehension readable and reducing visual noise.

Long Parameter List	`def f(a, b, c, d, e, f):`
	`  pass`
	
	`f(1, 2, 3, 4, 5, 6)`
		`args = (1, 2, 3, 4, 5, 6)`
		`f(*args)`
			Star in Function Call	✅	Defines and calls a function with six positional parameters, forcing every caller to remember order and count.	Packs values into one tuple and unpacks with *, making the call site shorter and signalling that the arguments travel as a bundle.

Long Method	`def process_data(data):`
	`  result = {}`
	`  for x in data:`
	`  if x:`
	`  x = x.strip().upper()`
	`  result[x] = len(x)`
	`  return result`
		`def process_data(data):`
		`  cleaned = [x.strip().upper() for x in data if x]`
		`  return {x: len(x) for x in cleaned}`
			List Comprehension, Dict Comprehension	✅	Cleans a list of strings and maps each to its length, but interleaves cleansing and accumulation in one loop.	Splits the concerns: a list comprehension does filtering/normalising, a dict comprehension builds the mapping, yielding shorter, testable code.

Complex Conditional	`if x > 10 and x < 50 and is_valid and not is_expired:` `  ...`	`if 10 < x < 50 and is_valid and not is_expired:` `  ...`	Chain Comparison, Truth-Value Test	✅	Evaluates bounds and flags in a verbose Boolean chain.	Chain comparison condenses the numeric test; relying on truthiness eliminates needless `== True/False` clutter.
Long Identifier	`total_number_of_successful_login = 12`	`total = 12`	Not Refactorable using Pythonic idioms	❌	Uses an overly wordy name for a simple scalar.	Replaces with a concise but meaningful identifier, improving scanability and maintenance.

Empty Catch Block	`try:`
	`  risky_operation()`
	`except:`
	`  pass`
		`try:`
		`  risky_operation()`
		`except Exception as e:`
		`  if e:`
		`  print(f"Error: {e}")`
			Truth-Value Test	✅	Silently ignores every exception, hiding failures.	Catches the specific base class, truth-tests the exception object, and logs it, surfacing problems without crashing.

Missing Default (match–case)	`match val:`
	`  case "a": ...`
	`  case "b": ...`
		`match val:`
		`  case "a": ...`
		`  case "b": ...`
		`  case _: handle_unknown()`
			Not Refactorable using Pythonic idioms	❌	Handles only two explicit cases, ignoring others.	Adds a wildcard branch so unexpected values are processed safely instead of falling through.

Long Lambda Function	`filter(`
	`  lambda user: len(user.posts) != 0,`
	`  users`
	`)`
		`[u for u in users if u.posts]`	Truth-Value Test, List Comprehension	✅	Anonymous lambda hides branching logic and forces a length check.	Uses truthiness (`if u.posts`) and a list comprehension to state intent directly, making the expression shorter and testable.

Long Message Chain	`user.get_profile().get_settings()`
	`.get_theme().get_color()`
		`profile, settings, theme = (`
		`  p := user.get_profile(),`
		`  s := p.get_settings(),`
		`  t := s.get_theme()`
		`)`
		`color = t.get_color()`
			Assign Multiple Targets	✅	Traverses four layers in one expression; a `None` anywhere raises late and obscures where it failed.	Binds each hop to a name in a single unpacking statement, surfaces errors sooner, and lets debuggers inspect each intermediate object.

Magic Number	`if age > 65:`	`SENIOR_AGE = 65` `if age > SENIOR_AGE:`	Not Refactorable using Pythonic idioms	❌	Uses a raw numeric literal with no context.	Extracts the constant into a named variable, documenting meaning and centralising future changes.

Complex Method	`def analyze(data):`
	`  result = {}`
	`  for item in data:`
	`  if isinstance(item, str):`
	`  if item:`
	`  cleaned = item.strip()`
	`  result[cleaned] = len(cleaned)`
	`  return result`
		`def analyze(data):`
		`  cleaned = [x.strip() for x in data if isinstance(x, str) and x]`
		`  return {x: len(x) for x in cleaned}`
			List Comprehension, Dict Comprehension	✅	Filters non-empty strings, strips them, and records their lengths with nested loops.	Comprehensions express filtering and mapping declaratively, producing shorter, clearer, and easily testable code.

where the targets field contains more than one variable (i.e., `len(node.targets) > 1`), as in the statement `a = b = 0`. Similarly, for the *Truth Value Test* idiom, it inspects `If` nodes and flags those whose condition (test) is not a `Compare` node, hence indicating implicit Boolean evaluation such as `if` items instead of `if len(items) > 0`. This idiom-specific logic is systematically applied across all Python files in a project, enabling large-scale and consistent detection of idiomatic usage patterns. The full list of supported idioms and their corresponding AST node patterns is shown in Table 6. To enable more accurate comparisons, we implemented the detection only for idioms detectable by RIdiom. For the sake of transparency and reproducibility, the tool is publicly available in our online appendix [10].

Before executing RIdiom against the considered software projects, we evaluated its actual detection capabilities. While the tool implements a set of heuristics grounded in the original *Pythonic* principles [2], we sought to ensure that these heuristics were correctly and consistently applied in practice. To this end, the first two authors jointly assessed a statistically significant random sample of the idiom instances detected by the tool across the dataset. Specifically, the validation was conducted over a sample size designed to achieve a 5% margin of error at a 95% confidence level. The goal of this manual inspection was to estimate the precision of the tool, that is, the proportion of correctly identified idioms among those flagged, thereby verifying whether the tool accurately implements the intended detection rules. We focused exclusively on precision, as estimating recall would have required a complete ground truth of all *Pythonic* idioms in the analyzed projects, which is not available. In fact, the lack of such a ground truth was a primary motivation for developing the custom detection tool in the first place. The results of this validation confirmed that the tool faithfully captures the idiomatic patterns it was designed to detect, corroborating the results obtained by the original authors [12] (accuracy = 100%) and supporting its use in ML systems.

To address the $\mathbf{RQ}_{1.2}$ and $\mathbf{RQ}_{1.3}$, we calculated both the percentage of *Pythonic* idioms and the *Pythonic* density in the source code as follows:

$$\text{Pythonic Perc.} = \frac{\#PythonicIdioms}{\#PythonicIdioms + \#Refactorable - PythonicIdioms} \tag{1}$$

Equation 1 calculates the *Pythonic* Percentage, which measures the share of idiomatic (*Pythonic*) code patterns relative to all idioms (both *Pythonic* and *Refactorable-Pythonic*) in the codebase. A higher value indicates a more idiomatic use of Python.

Finally, we computed the density as follows:

$$\text{(Refactorable-)Pythonic Density} = \frac{\#(Refactorable-)PythonicIdioms}{LOC} \tag{2}$$

Equation 2 computes the *(Refactorable-)Pythonic* Density, i.e., how often *Pythonic* or *Refactorable-Pythonic* appear per line of code. This helps us to assess how densely idiomatic patterns are used throughout the project.

Table 6: Idioms Extraction based on AST Nodes and Conditions.

Idiom	AST Node	Condition
Assign Multi Targets	`ast.Assign`	`len(node.targets) > 1`
Call Star	`ast.Call`	`isinstance(expr, ast.Starred)`
List Comprehension	`ast.ListComp`	Presence
Dict Comprehension	`ast.DictComp`	Presence
Set Comprehension	`ast.SetComp`	Presence
Truth Value Test	`ast.If`	`not isinstance(node.test, ast.Compare)`
Chain Compare	`ast.Compare`	`len(node.ops) > 1`
For Multiple Targets	`ast.For`	`isinstance(node.target, ast.List)`
Loop Else	`ast.For`	`node.orelse Presence`

*3.4. **RQ**$_2$: On the Relationship between Pythonic Idioms and Code Smells*

To answer **RQ**$_2$, we first checked the normality of the data to determine the most appropriate statistical test for our experiments. To assess the normality, we employed the *Shapiro-Wilk test* [39], which is particularly useful for small sample sizes and helps us understand whether our data is normally distributed. For each distribution, we split the data according to the NICHE dataset's "engineered" column, which distinguishes "well-engineered" from "non-engineered" projects - this distinction allowed us to address **RQ**$_{2.1}$ and **RQ**$_{2.2}$ separately using the same data analysis methods described in the following. Since the variables did not satisfy the normality assumption, we selected the Spearman test [40]—the non-parametric counterpart of Pearson [41]. We normalize data by dividing by NLOC. Furthermore, to mitigate the risk of false positives arising from multiple comparisons, we applied the Bonferroni correction to the resulting p-values [42]. We considered the p-value $\leqslant 0.05$ as statistically significant.

To perform our analysis, we considered the following dependent, independent, and control variables:

Dependent Variables: Since our objective is to evaluate the relationship between *(Refactorable)-Pythonic* idioms and code smells, we considered, as dependent variables, code smells detectable by DPy with a plausible mitigation strategy through *Pythonic* idioms. In particular, we thought the following Python-specific code smells: *Long Statement*, *Long Parameter List*, *Long Method*, *empty Catch Block*, *Complex Method*, *Complex Conditional*, *Long Lambda Function*, and *Long Message Chain*.

Independent Variable: As independent variables we selected all the nine *(Refactorable)-Pythonic* idioms detectable by RIdiom, *i.e., Assign Multiple Targets*, *Call Star*, *List Comprehension*, *Dict Comprehension*, *Set Comprehension*, *Truth Value Test*, *Chain Compare*, *For Multi Targets*, and *Loop Else*.

Control Variables: We selected the number of lines of code without comments (NLOC), the cyclomatic complexity, and the number of commits. The NLOC serves as an indicator of the codebase's size, a factor that naturally

correlates with the likelihood of encountering code smells [7]. Cyclomatic complexity reflects the structural intricacy of the code and can influence the presence of code smells independently of the proportion of Python code. Finally, the number of commits captures the overall development and maintenance activity within a project's history, offering insight into how actively the codebase has evolved. We extracted these metrics using SonarQube [43].

It is important to note that, on the one hand, we performed the same analysis considering *Pythonic* and *Refactorable-Pythonic* idioms; on the other hand, we consider only these metrics as control variables due to the lack of useful tools for extracting other candidate control variables.

4. Analysis of the Results

Before reporting and discussing our results, we provide a preliminary statistical description of the NICHE dataset of code smell diffusion.

Table 7: Descriptive statistics of code smells in well-engineered and not well-engineered projects.

Group	Mean	Std. Dev.	Median	Min	Max
Well-Engineered	785.790	587.650	616	9	2,782
Non-engineered	459.270	499.090	284	5	2,576

Table 7 presents descriptive statistics on Python-specific code smells, comparing "well-engineered" and "non-engineered" projects. On average, "well-engineered" projects exhibit a higher number of Python-specific smells (mean = 785.790) compared to "non-engineered" projects (mean = 459.270). The standard deviation values are also substantial in both groups, indicating considerable variability in smell counts across projects. These findings may suggest that "well-engineered" projects tend to contain more code overall—or that smell detection is more thorough in such projects—resulting in higher raw counts of Python-specific code smells.

To reinforce our analysis, we verified whether there were statistically significant differences between "well-engineered" and "non-engineered" projects in terms of smell distribution. In-depth, we normalized for KLOC for each smell distribution, then checked normality using Shapiro-Wilk [44] test. Since the distributions do not respect the normality assumption, we selected the Mann-Whitney U test [45], a non-parametric version of the T-Test [46]—our results discovered no statistical differences between distributions.

4.1. $\boldsymbol{RQ}_{1.1}$: On the Frequencies of Idioms in ML Projects

Figure 2 displays the frequencies of *Pythonic* (gray bar) and *Refactorable-Pythonic* (red bar) idioms. Among *Pythonic* idioms, the most frequent is the *Truth Value Test*, followed by *List Comprehension*, and *For Multiple Targets*. These idioms enable concise and readable code, particularly when dealing with loops or complex data manipulations. In addition, among the "comprehension" constructs, *List Comprehension* is the most widely used. This result corroborates previous ones [3], identifying *List Comprehension* as one of the most adopted idioms in the Python codebase.

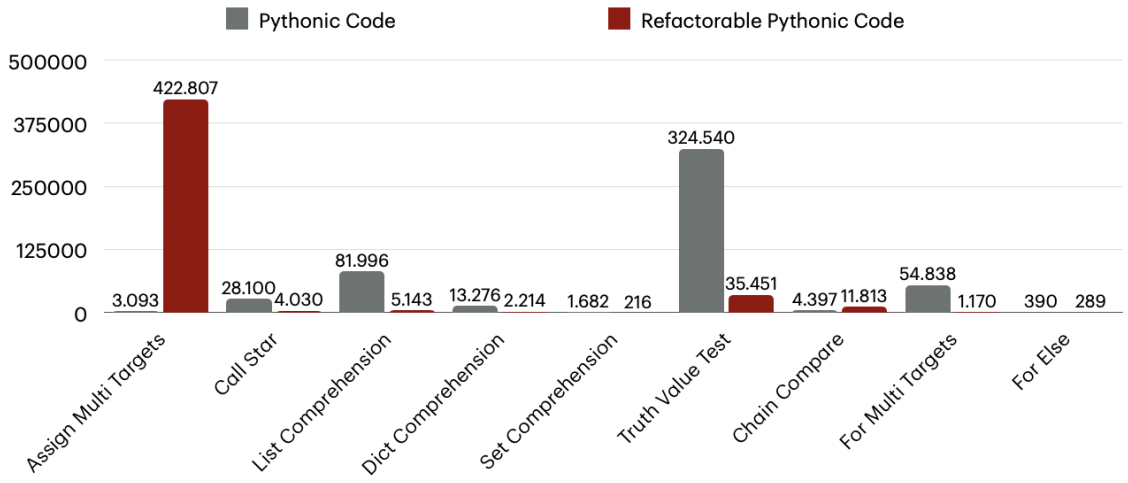

Figure 2: Frequencies of Pythonic and Refactorable-Pythonic Idioms.

Moving on to *Refactorable-Pythonic* idioms, the most frequent idiom observed is *Assign Multiple Targets*, followed by *Truth Value Test* and *Chain Compare*. These results could suggest that practitioners may often write multiple assignment statements separately, rather than utilizing Python's ability to assign multiple variables in a single statement. In both *Pythonic* and *Refactorable-Pythonic* idioms, the less frequent idiom is *For Else*, suggesting the unpopularity of this construct in ML projects.

$RQ_{1.1}$ Summary

The most frequent idioms are *Truth Value Test* and *Assign Multi Targets* for *Pythonic* and *Refactorable-Pythonic* idioms, respectively, while the least frequent is the *For Else* idiom.

4.2. $\boldsymbol{RQ_{1.2}}$*: On the Variation of Project Size*

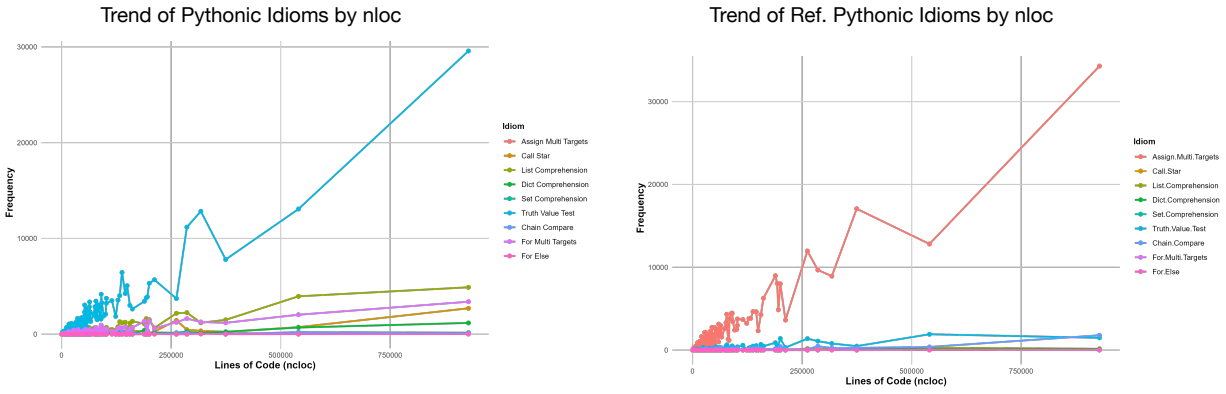

Figure 3: Variation of Pythonic Idioms and Non-Pythonic Idioms with Respect to Project Size.

Figure 3 illustrates the variation in the usage of *Pythonic* idioms (left side) and *Refactorable-Pythonic* idioms

(right side) with respect to the project size (in NLOC). As shown in the figure, the most frequently adopted *Pythonic* idiom is the *Truth Value Test*. In contrast, the most prevalent *Refactorable-Pythonic*, refactorable idiom is *Assign Multiple Targets*. In both cases, it is possible to observe that the growth of the *Truth Value Test* and *Assign Multiple Targets* idioms accelerates dramatically after 250,000 lines. We observed that as the project size increases, both *Pythonic* and *Refactorable-Pythonic* idioms increase. The substantial rise in *Truth Value Test* indicates that *Pythonic* best practices become increasingly essential as the complexity of the codebase grows. Meanwhile, the persistent use of *Assign Multi Targets* highlights the presence of Refactorable-idiomatic patterns prevalent in larger projects, underscoring the need for targeted refactoring efforts.

RQ$_{1.2}$ Summary

If we normalize by project size, the most prevalent *Pythonic* is *Truth Value Test*, and the most prevalent *Refactorable-Pythonic* idiom is *Assign Multiple Targets*.

4.3. ***RQ***1_3*: On the Density of Idioms*

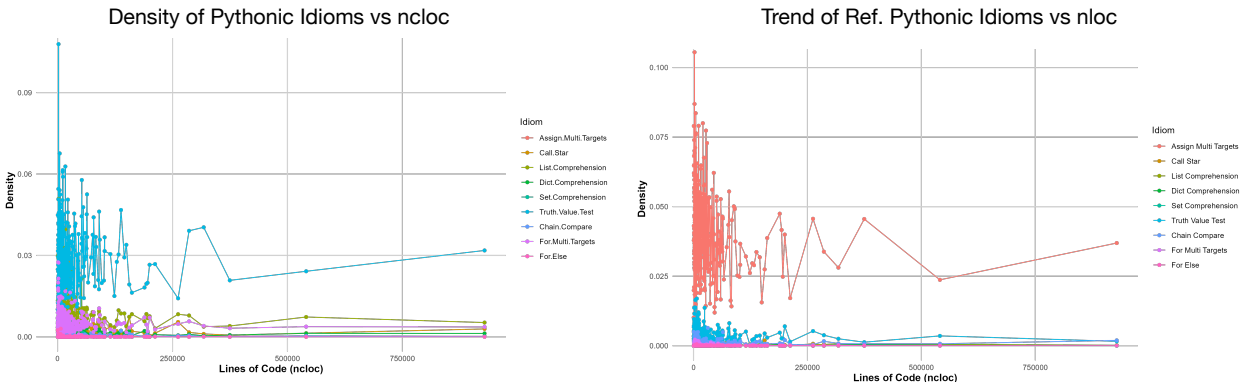

Figure 4: Density of Pythonic and Refactoraizable-Pythonic Idioms.

Figure 4 presents the density distribution of idioms relative to NLOC, separately for *Pythonic* idioms (left) and *Refactorable-Pythonic* idioms (right). Among the idioms considered, *Truth Value Test* emerges as the most frequently used in the Pythonic category, while *Assign Multiple Targets* is the most prevalent among the refactorable subset. These findings align with the observations made in RQ$_{1.2}$, where both idioms also appeared as prominent in terms of frequency.

The high density of *Truth Value Test* likely reflects its widespread use as a concise and idiomatic pattern for evaluating conditions—an operation that is both common and semantically lightweight. Its idiomaticity and simplicity may explain its frequent appearance even in projects with varying levels of engineering discipline. Similarly, *Assign Multiple Targets* is syntactically compact and often used to streamline code, which may contribute to its diffusion across projects regardless of quality.

$RQ_{1.3}$ Summary

The most adopted *Pythonic* idiom in terms of density is *Truth Value Test*, while the most Refactorable idiom is *Assign Multi Targets*.

4.4. RQ_2: On the Relationship Between Idioms and Smells

Before analyzing the differences observed between "well-engineered" and "non-engineered" projects, let us provide an overall overview of the correlation between the presence of idioms and code smells. Table 8 shows the results of our statistical test, with p-values adjusted with Bonferroni correction [42]. We can observe that, even after normalizing, certain *Refactorable-Pythonic* idioms (*e.g., Call Star*, *Chain Compare*) maintain moderate positive correlations ($\rho \approx 0.30$–0.36) with Python-specific code smells (such as *Complex Conditional*, and *Long Parameter List*). *Pythonic* constructs like *Truth Value Test* also appear, but with smaller effect sizes ($\rho \approx 0.23$–0.26). In contrast, control variables, especially *NLOC*, emerge as dominant factors. For instance, $\rho(\text{NLOC}, \text{Long Statement}) = 0.73$ and $\rho(\text{NLOC}, \text{Long Method}) = 0.69$, indicating that file size is the strongest predictor of code smell incidence. Additionally, *File Complexity* correlates with *Complex Method* at $\rho = 0.37$ ($p < 10^{-11}$, ***), and *Commits* also shows moderate correlations with several smells. Overall, these results indicate that while idiom usage, particularly of *Refactorable-Pythonic* constructs, does relate to certain code smells, structural factors such as file size and complexity are more strongly associated with smell incidence. This provides a baseline understanding for interpreting the role of engineering practices in the patterns observed. In the remainder of this section, we turn to the analysis of the results that directly addresses $\mathbf{RQ}_{2.1}$ and $\mathbf{RQ}_{2.2}$.

Table 8: Spearman Correlation between *(Refactorable-)Pythonic* Idioms and Control Variables versus Code Smells (normalized by NLOC), with Bonferroni-adjusted significance levels

Idiom	Smell	Spearman_ρ	p-value	Significance
Refactorable-Pythonic Chain Compare	Complex Conditional	0.35	2.92e-10	***
Refactorable-Pythonic Call Star	Long Parameter List	0.34	1.86e-09	***
Assign Multi Targets	Long Parameter List	0.30	7.30e-08	***
Refactorable-Pythonic Chain Compare	Complex Method	0.30	1.23e-07	***
Refactorable-Pythonic Call Star	Complex Method	0.30	1.35e-07	***
Refactorable-Pythonic Call Star	Long Method	0.28	9.90e-07	***
Refactorable-Pythonic Call Star	Complex Conditional	0.26	3.92e-06	**
Refactorable-Pythonic List Compreh.	Complex Method	0.26	4.79e-06	**
Truth Value Test	Complex Method	0.25	1.17e-05	**
Refactorable-Pythonic Chain Compare	Long Parameter List	0.25	1.47e-05	**
Set Comprehension	Empty Catch Block	0.24	2.50e-05	**
Refactorable-Pythonic Chain Compare	Long Method	0.24	2.73e-05	**

Continued on next page

Idiom	Smell	Spearman$_\rho$	p-value	Significance
Assign Multi Targets	Complex Method	0.23	5.20e-05	*
Refactorable-Pythonic Call Star	Long Statement	0.23	5.66e-05	*
Truth Value Test	Complex Conditional	0.23	5.85e-05	*
Truth Value Test	Empty Catch Block	0.23	6.54e-05	*
Chain Compare	Complex Method	0.22	8.08e-05	*
Assign Multi Targets	Long Method	0.22	9.70e-05	*
Control Variable	**Smell**	**Spearman$_\rho$**	**p-value**	**Significance**
NLOC	Long Statement	0.73	1.64e-51	***
NLOC	Long Method	0.69	1.62e-44	***
NLOC	Complex Method	0.65	1.75e-38	***
NLOC	Long Parameter List	0.61	1.16e-31	***
NLOC	Complex Conditional	0.41	7.72e-14	***
NLOC	Long Lambda Function	0.31	2.31e-08	***
NLOC	Long Message Chain	0.25	1.36e-05	*
File Complexity	Complex Method	0.37	1.65e-11	***
File Complexity	Complex Conditional	0.32	1.09e-07	***
File Complexity	Long Method	0.31	2.54e-08	***
File Complexity	Long Parameter List	0.27	2.27e-06	***
Commits	Long Statement	0.29	2.49e-07	***
Commits	Complex Method	0.27	1.44e-06	***
Commits	Empty Catch Block	0.24	1.98e-05	*
Commits	Long Method	0.22	8.43e-05	*

$^*\ p < 0.1$; $^{**}\ p < 0.05$; $^{***}\ p < 0.01$

*4.4.1. **RQ**$_{2.1}$: On the Correlation between Idioms and Code Smells in Well-Engineered Projects*

Table 9: Spearman Correlation between *(Refactorable-)Pythonic* Idioms and Control Variables versus Code Smells for "Well-Engineered" Projects (normalized by NLOC), with Bonferroni-corrected significance levels

Idiom	Smell	Spearman$_\rho$	p-value	Significance
Chain Compare	Complex Conditional	0.35	4.48e-08	***
Call Star	Long Parameter List	0.33	5.01e-07	***
Chain Compare	Complex Method	0.31	1.78e-06	***
Call Star	Complex Method	0.29	6.26e-06	**

Continued on next page

Idiom	Smell	Spearman$_\rho$	p-value	Significance
Call Star	Complex Conditional	0.27	2.94e-05	**
Chain Compare	Long Parameter List	0.26	7.31e-05	*
Truth Value Test	Complex Method	0.26	6.54e-05	*
Call Star	Long Method	0.26	9.46e-05	*
Control Variable	**Smell**	**Spearman$_\rho$**	**p-value**	**Significance**
NLOC	Long Statement	0.71	4.49e-13	***
NLOC	Long Method	0.69	2.91e-11	***
NLOC	Complex Method	0.64	9.38e-10	***
NLOC	Long Parameter List	0.61	5.39e-08	***
NLOC	Complex Conditional	0.39	1.21e-04	***
File Complexity	Complex Method	0.48	1.19e-05	**
File Complexity	Long Method	0.31	2.50e-06	***
File Complexity	Long Parameter List	0.25	1.06e-04	*

$^*\ p < 0.1$; $^{**}\ p < 0.05$; $^{***}\ p < 0.01$

Restricting the analysis to "well-engineered" projects (Table 9), we observe that the same idioms, i.e., particularly *Call Star* and *Chain Compare*, still correlate moderately with Python-specific smells such as *Complex Conditional* and *Long Parameter List* ($\rho \approx 0.30$–0.36). This suggests that the mere presence of engineering practices (e.g., tests, CI, structured packaging) does not eliminate the co-occurrence of idioms and smells, especially for constructs that inherently increase syntactic or semantic complexity. Interestingly, *Pythonic* idioms show weaker associations overall, reinforcing the intuition that idioms aligned with Python's idiomatic style may be less likely to contribute to maintainability issues, even in well-structured repositories. Such a correlation analysis seems to reinforce the value of idiomatic style, while also highlighting that some idioms, when overused or applied without sufficient caution, may correlate with specific smells regardless of engineering discipline. However, the strongest correlations remain associated with control variables, particularly structural complexity indicators. *NLOC* continues to be the most dominant factor, with $\rho(\text{NLOC} = 0.71)$ for *Long Statement* and $\rho(\text{NLOC} = 0.69)$ for *Long Method*, showing that larger files are more prone to certain smells, even in projects with sound engineering processes. *File Complexity* and *Commits* also show non-negligible correlations, textitasizing the multifaceted nature of smell emergence. These findings suggest that while engineering practices improve overall quality, they do not fully mitigate the stylistic or structural risks associated with specific idiom usage. Therefore, effective engineering should combine process-oriented practices with style-aware and complexity-aware development guidelines.

RQ$_{2.1}$ Summary

Considering "well-engineered" projects, we observe that some idioms (*e.g., Call Star* and *Chain Compare*) positively correlate with the presence of smells. Our results also confirm the correlation between NLOC and file complexity with some smells.

4.4.2. $\boldsymbol{RQ}_{2.2}$: *On the Correlation between Idioms and Code Smells in non-engineered Projects*

Table 10: Spearman Correlation between *(Refactorable-)Pythonic* Idioms and Control Variables versus Code Smells for "Not-Engineered" Projects (normalized by NLOC), with Bonferroni-corrected significance levels

Control Variable	Smell	Spearman$_\rho$	p-value	Significance
NLOC	Long Statement	0.71	4.49e-13	***
NLOC	Long Method	0.67	2.91e-11	***
NLOC	Long Parameter List	0.64	4.53e-10	***
NLOC	Complex Method	0.63	8.29e-10	***
File Complexity	Complex Method	0.48	1.19e-05	**

$^{*}\, p < 0.1$; $^{**}\, p < 0.05$; $^{***}\, p < 0.01$

The results for "non-engineered" projects, reported in Table 10, show a markedly different profile compared to their well-engineered counterparts. Here, control variables overwhelmingly dominate the top correlations. In particular, we noticed: $\rho(NLOC, \textit{Long Statement}) = 0.71$, $\rho(NLOC, \textit{Long Method}) = 0.67$, and $\rho(NLOC, \textit{Long Parameter List}) = 0.64$. These strong associations seems to confirm that, in the absence of structured engineering practices, code size alone becomes the primary driver of code quality degradation. Moreover, *File Complexity* also shows a substantial correlation with *Complex Method* ($\rho = 0.48$, **), reinforcing the idea that cyclomatic or structural complexity further compounds the likelihood of smell incidence. Interestingly, no statistically significant correlations were found between idioms, either *Pythonic* or *Refactorable-Pythonic*, and code smells in this group. This absence suggests that, in poorly governed codebases, the stylistic layer provided by idioms is largely irrelevant in the face of overwhelming structural issues. This stark contrast with the "well-engineered" subset highlights an important insight: idiom–smell associations may only become visible or meaningful when basic quality controls are in place. In other words, engineering practices seems to act as a kind of baseline hygiene, enabling finer-grained influences such as idiom usage to surface. Without such a foundation, the signal from idioms is effectively drowned out by noise from unchecked size and complexity. These findings support the view that improving software quality in low-discipline environments requires first addressing fundamental structural issues, such as limiting file size and managing complexity, before style-related interventions (like idiom use) can have measurable impact.

RQ$_{2.2}$ Summary

In "non-engineered" projects, code smells are most strongly associated with structural control variables rather than idiomatic usage. *NLOC* shows the highest correlations, particularly with *Long Statement* ($\rho = 0.71$), *Long Method* ($\rho = 0.67$), and *Long Parameter List* ($\rho = 0.64$). *File Complexity* also correlates with *Complex Method* ($\rho = 0.48$). No significant correlations involving idioms were found, suggesting that in the absence of structured engineering practices, large-scale complexity metrics are the primary indicators of code smells.

5. Discussion and Implications

The findings of our empirical investigation are statistical in nature and, as such, reveal correlations, rather than causation, between the adoption of *Pythonic* idioms and code quality. Nonetheless, these correlations can still provide meaningful insights for both practitioners and researchers, potentially informing further analyses aimed at uncovering the underlying causes behind these relationships.

From $\mathbf{RQ}_1$, we observe that idioms such as *Truth Value Test* and *Assign Multiple Targets* are among the most frequently used across ML projects—both in absolute frequency and density. However, $\mathbf{RQ}_2$ shows that these idioms exhibit moderate to weak positive correlations with various code smells. Specifically, *Truth Value Test* correlates with *Complex Method*, and weakly with *Complex Conditional*. Similarly, *Assign Multiple Targets* shows moderate correlation with *Long Parameter List* and weak correlations with *Long Method*, and *Complex Method*.

These findings suggest that even idioms typically intended to improve readability and maintainability may still be present in code that exhibits structural or semantic issues, which themselves might reflect even graver underlying software conditions. This applies not only to idioms present in the original code but also to those introduced via refactoring from *Refactorable-Pythonic* patterns. This contradiction is particularly evident in well-engineered ML projects, where *Pythonic* idioms are more frequently adopted yet still coexist with a high prevalence of Python-specific code smells. Such a pattern suggests that *Pythonic* idioms, while syntactically elegant, may obscure deeper structural complexities when applied without careful consideration.

This consideration would therefore require further experimentation to determine its consequences. While *Pythonic* idioms offer expressive and concise syntax, our results show that their presence does not necessarily imply higher code quality. In "well-engineered" ML projects, idioms often co-occur with code smells, indicating that idiomatic style alone is insufficient to ensure maintainability. These constructs may be used to streamline complex logic, but when applied without attention to structural design, they can obscure issues rather than clarify them. This highlights the importance of considering idioms as part of a broader strategy for managing complexity. Developers should assess whether idiomatic constructs actually enhance clarity and modularity in their specific context, rather than applying them blindly for stylistic compliance. Similarly, linters and refactoring tools should avoid one-size-fits-all suggestions and instead provide context-aware recommendations that account for complexity metrics and maintainability goals. Although our study does not directly measure contributors' backgrounds or intent, the variation in idiom–smell associations between "well-engineered" and "non-engineered" projects suggests that idioms tend to have more measurable impact on code quality only when sound engineering practices are already in place. This underlines the need for tools that combine idiom detection with quality indicators, offering smarter feedback tailored to the structure and scale of ML systems.

Finally, in $\mathbf{RQ}_2$, we observed a notable presence of code smells in "well-engineered" projects. At first glance, this might appear contradictory, as one would expect such systems to exhibit cleaner code. However, this trend can be interpreted looking at key structural differences between "well-engineered" and "non-engineered" projects. The former systems are often larger, longer-lived, and more actively maintained - as shown by the statistics about the median commits and NLOC of the projects across the two groups (Table 2). As such, they are more likely to accumulate complexity and technical debt over time, despite following sound engineering practices. Their extensive

size and functionality may also expose a wider surface for smell detection, especially when idioms are used to streamline complex logic. In contrast, the latter are typically smaller, less mature, or less actively maintained, and may contain simpler or more static codebases. While they may lack formal practices or idiomatic consistency, their limited scope often results in lower detectable complexity and fewer opportunities for smells to emerge.

As a final consideration, the prevalence of idioms in "well-engineered" projects seems to suggest that their usage is often part of an effort to simplify complex logic. However, without complementary architectural practices, such as systematic refactoring, modularization, or decomposition, the use of *Pythonic* idioms may only offer shallow relief. It follows that idiomatic constructs should be considered as part of a broader strategy for managing complexity, rather than a standalone indicator of clean code.

On the basis of the findings of the study and subsequent considerations coming from them, we may finally address the overarching question of the study asked in Section 1, i.e., whether the use of *Pythonic* idioms correlate with higher or lower code quality in real-world ML codebases: applying idioms or refactoring to *Pythonic* style does not guarantee cleaner or more maintainable code. While idioms promote readability and conciseness, their misapplication may introduce or obscure structural issues. The effectiveness of idioms may be influenced by the developers' background, project size, and architectural practices. Therefore, *Pythonic* idioms should be considered complementary tools rather than guarantees of quality, requiring careful and informed use supported by context-sensitive development tools.

6. Threats to Validity

This section discusses threats that might have affected the findings of our study, and how we mitigated them.

Construct Validity. This category concerns the potential discrepancies between theory and observation. Dataset selection plays a crucial role, and using an established dataset from prior literature helps mitigate bias from uncontrolled variables. Since our study centers on the concept of *Pythonic* code, rooted in the Python programming community, we focused on projects predominantly written in Python. Additionally, tool selection is critical, as different tools yield different metrics.

For code smell extraction, we used an already validated tool *i.e.,* we selected DPy for Pythonic-specific code smells. In addition, we used SonarQube to extract additional metrics, *i.e.,* NLOC, file complexity, and number of commits. To maintain focus on *Pythonic* elements, we analyzed only Python files in projects where more than 50% of the code was in Python. This approach helps avoid noise from *Refactorable-Pythonic* sources, such as code written in other languages. To identify Refactorable-idiomatic Python code, we used the RIdiom tool. The *Pythonic* variables in the study (*Density, Percentage*, *Count*) were extracted through AST traversing and validated with manual testing to ensure alignment with the nine idioms defined in the RIdiom paper [12]. While *Pythonic* code is not limited to these idioms, future studies incorporating additional idioms could enrich our findings.

Internal Validity. These threats relate to factors that might have influenced the study's results. In $\mathbf{RQ}_2$, we examined the relationship between *Pythonic* idioms and code quality, measured via code smells. While collecting data through DPy, we also gathered metrics on project size, number of files, and cyclomatic complexity. These

were used as control variables alongside the *Pythonic* metrics to reduce confounding effects. Last but not least, it is important to note that our study does not aim to claim causation between the presence (or not) of *Pythonic* idioms and code quality, but, rather, at showing correlations.

Conclusion Validity. Conclusion Validity threats pertain to the choice and use of statistical tests. Before applying statistical tests (*e.g.,* t-tests, Wilcoxon tests) and models, we ensured that the data met the necessary assumptions. We set a significance level of 0.05, which was consistently respected across all tests and models. While our study is quantitative and serves as a preliminary exploration, further qualitative research is needed to understand how *Pythonic* idioms impact code quality.

External Validity. The primary threat to external validity stems from the dataset used. The selection of projects is crucial, so we selected the NICHE dataset, a large collection of 303 real-world ML-enabled systems.These projects vary in context, size, and number of files, supporting the generalization of our findings to open-source ML projects with similar characteristics. Although the results may not directly apply to industrial projects, we encourage further research to replicate our study on closed-source projects, allowing for comparisons between different settings. Another potential threat is the selection of *Pythonic* idioms. Although *Pythonic* code encompasses many idioms and conventions, RIdiom is the only tool that can identify Refactorable-idiomatic Python code. Our focus on nine specific idioms reflects a balance between identifying commonly used idioms and detecting those underutilized but potentially beneficial. While this limits our definition of *Pythonic* to these idioms, future studies employing additional idioms or other coding practices could provide a more comprehensive view.

7. Conclusions

This paper proposed an empirical investigation into the relationship between *Pythonic* and *Refactorable-Pythonic* idioms and code smells in ML projects. We analyzed 303 Python projects classified as "well-engineered" or "non-engineered", discovering that the more frequent *Pythonic* idioms are *Truth Value Test* and *Assign Multi Target* for "well-engineered" and "non-engineered", respectively.

Our statistical analyses revealed a correlation between the usage of *(Refactorable-)Pythonic* idioms and code smells, indicating that refactoring *Refactorable-Pythonic* code into *Pythonic* idioms does not consistently correspond with improved code quality in ML systems. Indeed, in a non-negligible number of cases, *Pythonic* idioms correlated with some code smells, especially in "well-engineered" projects. We conclude our study by providing implications and discussions to drive further research on the matter.

As future work, we plan to extend our investigation by incorporating a broader range of *Pythonic* idioms beyond the nine currently studied, potentially capturing a more comprehensive picture of idiomatic usage in ML codebases. Furthermore, we aim to complement our quantitative analysis with qualitative methods to better understand the rationale behind using certain idioms and their perceived impact on maintainability. Finally, another promising direction involves exploring the interplay between idiomatic usage and other quality attributes beyond code smells, such as performance or testability.

Acknowledgment

This work has been partially supported by the *Qual-AI* national research project, funded by the MUR under the PRIN 2022 programs (Code: D53D23008570006).

Declaration of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The manuscript includes data as electronic supplementary material. In particular, datasets, detailed results, as well as scripts and additional resources useful for reproducing the study, are available as part of our online appendix on Figshare available at: [10] and at the following GitHub page: `https://giammariagiordano.github.io/pythonic_and_smells/index.html`

Credits

Gerardo Festa: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Giammaria Giordano:** Conceptualization, Methodology, Validation, Supervision, Resources, Writing - Original Draft & Editing. **Valeria Pontillo:** Conceptualization, Methodology, Validation, Writing - Review & Editing. **Massimiliano Di Penta:** Writing - Review & Editing. **Damian A. Tamburri:** Resources, Writing - Review & Editing. **Fabio Palomba:** Supervision, Resources, Writing - Review & Editing.

References

[1] I. Karamitsos, S. Albarhami, C. Apostolopoulos, Applying devops practices of continuous automation for machine learning, Information 11 (2020) 363.

[2] T. Peters, The zen of python, PEP 20, `https://www.python.org/dev/peps/pep-0020/`, 2004. Accessed: October 6, 2024.

[3] C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall, G. Robles, 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, 2018.

[4] C. Zid, F. Zampetti, G. Antoniol, M. Di Penta, A study on the pythonic functional constructs' understandability, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[5] G. Busquim, H. Villamizar, M. J. Lima, M. Kalinowski, On the interaction between software engineers and data scientists when building machine learning-enabled systems, in: International Conference on Software Quality, Springer, 2024, pp. 55–75.

[6] G. Annunziata, S. Lambiase, D. A. Tamburri, W.-J. van den Heuvel, F. Palomba, G. Catolino, F. Ferrucci, A. De Lucia, Uncovering community smells in machine learning-enabled systems: Causes, effects, and mitigation strategies, ACM Transactions on Software Engineering and Methodology (2024).

[7] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[8] A. Boloori, T. Sharma, Dpy: Code smells detection tool for python (2025) 826–830.

[9] R. Widyasari, Z. Yang, F. Thung, S. Q. Sim, F. Wee, C. Lok, J. Phan, H. Qi, C. Tan, Q. Tay, et al., Niche: A curated dataset of engineered machine learning projects in python, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), IEEE, 2023, pp. 62–66.

[10] G. Festa, G. Giammaria, P. Valeria, D. P. Max, T. Damian, P. Fabio, Pythonic vs refactorable pythonic: On the relationship between pythonic idioms and code quality in machine learning projects, 2025. URL: https://zenodo.org/records/16537820.

[11] A. Martelli, A. Ravenscroft, S. Holden, Python in a Nutshell: A desktop quick reference, " O'Reilly Media, Inc.", 2017.

[12] Z. Zhang, Z. Xing, X. Xu, L. Zhu, Ridiom: Automatically refactoring non-idiomatic python code with pythonic idioms, in: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2023, pp. 102–106. doi:10.1109/ICSE-Companion58688.2023.00034.

[13] Z. Chen, L. Chen, W. Ma, B. Xu, Detecting code smells in python programs, in: 2016 International Conference on Software Analysis, Testing and Evolution (SATE), 2016, pp. 18–23. doi:10.1109/SATE.2016.10.

[14] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change-and fault-proneness, Empirical Software Engineering 17 (2012) 243–275.

[15] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, 2018.

[16] G. Giordano, A. Fasulo, G. Catolino, F. Palomba, F. Ferrucci, C. Gravino, On the evolution of inheritance and delegation mechanisms and their impact on code quality, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 947–958.

[17] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (2017) 1063–1088.

[18] G. Giordano, G. Sellitto, A. Sepe, F. Palomba, F. Ferrucci, The yin and yang of software quality: On the relationship between design patterns and code smells, in: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2023.

[19] F. Pecorelli, F. Palomba, D. Di Nucci, A. De Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, in: 2019 IEEE/ACM 27th international conference on program comprehension (ICPC), IEEE, 2019, pp. 93–104.

[20] F. Pecorelli, F. Palomba, F. Khomh, A. De Lucia, Developer-driven code smell prioritization, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 220–231.

[21] M. De Stefano, F. Pecorelli, F. Palomba, A. De Lucia, Comparing within-and cross-project machine learning algorithms for code smell detection, in: Proceedings of the 5th international workshop on machine learning techniques for software quality evolution, 2021, pp. 1–6.

[22] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 268–278.

[23] N. Vavrová, V. Zaytsev, Does python smell like java? tool support for design defect discovery in python, arXiv preprint arXiv:1703.10882 (2017).

[24] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, B. Xu, Understanding metric-based detectable smells in python software: A comparative study, Information and Software Technology 94 (2018) 14–29.

[25] B. Van Oort, L. Cruz, M. Aniche, A. Van Deursen, The prevalence of code smells in machine learning projects, in: 2021

IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN), IEEE, 2021, pp. 1–8.

[26] H. Jebnoun, H. Ben Braiek, M. M. Rahman, F. Khomh, The scent of deep learning code: An empirical study, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020.

[27] G. Giordano, A. Della Porta, F. Ferrucci, F. Palomba, An evidence-based study on the relationship of software engineering practices on code smells in python ml projects, in: 2025 51st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2025.

[28] F. Zampetti, F. Belias, C. Zid, G. Antoniol, M. Di Penta, An empirical study on the fault-inducing effect of functional constructs in python, in: IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022, 2022, pp. 47–58. URL: https://doi.org/10.1109/ICSME55016.2022.00013. doi:10.1109/ICSME55016.2022.00013.

[29] F. Zampetti, C. Zid, G. Antoniol, M. Di Penta, The downside of functional constructs: a quantitative and qualitative analysis of their fix-inducing effects, Empir. Softw. Eng. 30 (2025) 9. URL: https://doi.org/10.1007/s10664-024-10568-z. doi:10.1007/S10664-024-10568-Z.

[30] T. Sakulniwat, R. G. Kula, C. Ragkhitwetsagul, M. Choetkiertikul, T. Sunetnanta, D. Wang, T. Ishio, K. Matsumoto, Visualizing the usage of pythonic idioms over time: A case study of the with open idiom, in: 2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP), 2019, pp. 43–435. doi:10.1109/IWESEP49350.2019.00016.

[31] P. Phan-Udom, N. Wattanakul, T. Sakulniwat, C. Ragkhitwetsagul, T. Sunetnanta, M. Choetkiertikul, R. G. Kula, Teddy: automatic recommendation of pythonic idiom usage for pull-based software projects, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 806–809.

[32] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, Making Python code idiomatic by automatic refactoring non-idiomatic python code with Pythonic idioms, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 696–708.

[33] P. Leelaprute, B. Chinthanet, S. Wattanakriengkrai, R. G. Kula, P. Jaisri, T. Ishio, Does coding in pythonic zen peak performance? preliminary experiments of nine pythonic idioms at scale, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 575–579.

[34] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, Q. Lu, Faster or slower? performance mystery of python idioms unveiled with empirical evidence, in: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, IEEE, 2023, pp. 1495–1507.

[35] C. Zid, F. Belias, M. Di Penta, F. Khomh, G. Antoniol, List comprehension versus for loops performance in real python projects: Should we care?, in: IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024, 2024, pp. 592–601. URL: https://doi.org/10.1109/SANER60148.2024.00066. doi:10.1109/SANER60148.2024.00066.

[36] A. Nagpal, G. Gabrani, Python for data analytics, scientific and technical applications, in: 2019 Amity International Conference on Artificial Intelligence (AICAI), 2019, pp. 140–145. doi:10.1109/AICAI.2019.8701341.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, et al., Experimentation in software engineering, Springer, 2012.

[38] J. Tan, M. Lungu, P. Avgeriou, Towards studying the evolution of technical debt in the python projects from the apache software ecosystem., in: BENEVOL, 2018, pp. 43–45.

[39] Z. Hanusz, J. Tarasinska, W. Zielinski, Shapiro–wilk test with known mean, REVSTAT-Statistical Journal 14 (2016) 89–100.

[40] J. C. De Winter, S. D. Gosling, J. Potter, Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data., Psychological methods 21 (2016) 273.

[41] P. Sedgwick, Pearson's correlation coefficient, Bmj 345 (2012).

[42] E. W. Weisstein, Bonferroni correction, https://mathworld. wolfram. com/ (2004).

[43] G. A. Campbell, P. P. Papapetrou, SonarQube in action, Manning Publications Co., 2013.

[44] N. M. Razali, Y. B. Wah, et al., Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests, Journal of statistical modeling and analytics 2 (2011) 21–33.

[45] P. E. McKnight, J. Najab, Mann-whitney u test, The Corsini encyclopedia of psychology (2010) 1–1.

[46] T. K. Kim, T test as a parametric statistic, Korean journal of anesthesiology 68 (2015) 540–546.