Pythonic vs Not Pythonic: On the Relationship between Pythonic Idioms and Code Quality in Machine Learning Projects

Author 1,[1] Author 2,[2]

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

Corresponding author:

Pythonic vs Not Pythonic: On the Relationship between Pythonic Idioms and Code Quality in Machine Learning Projects

Author 1,[1] Author 2,[2]

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

Abstract

Python is increasingly becoming the *lingua franca* for developing Machine Learning (ML) systems, thanks to its rich ecosystem of libraries and its emphasis on readability and simplicity. In this context, so-called *Pythonic* idioms are commonly viewed as stylistic conventions that may support maintainable and efficient code when used appropriately. Conversely, *non-Pythonic* idioms refer to code constructs that, while syntactically correct, can be refactored into more idiomatic and maintainable Python code. Despite this common assumption, empirical evidence linking these idiomatic patterns to software quality in ML projects remains scarce.

This paper aims to fill this gap by empirically investigating the impact of both Pythonic and non-Pythonic idioms on code quality. To this end, we analyze 303 open-source Python projects, categorized as "well-engineered" and "not-engineered", using DPy to detect Pythonic code smells, CodeSmile for ML-specific code smells, and SonarQube to extract bug-related data.

Our results show that the "*Truth Value Test*" is the most commonly used Pythonic idiom, while "*Assign Multi Targets*" is the most prevalent non-Pythonic idiom. In "well-engineered" projects, we find statistically significant correlations between both Pythonic and non-Pythonic idioms and the presence of code smells. In contrast, "not-engineered" projects show significant relationships primarily with structural metrics such as lines of code, number of commits, and file complexity. Additionally, our analysis reveals statistically significant correlations between idioms and bugs in both project categories, suggesting that the adoption of Pythonic idioms does not always guarantee better code quality in ML systems.

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

1. Introduction

The rapid growth of Machine Learning (ML) has led to an increased focus on developing code that is not only functional but also readable, maintainable, and scalable. Python—as the predominant language in ML development—encourages the use of *Pythonic* idioms, that is, coding practices that leverage Python's expressive syntax for simplicity, clarity, and efficiency [1]. While Pythonic idioms are commonly viewed as stylistic patterns that may contribute to more maintainable and efficient code when appropriately applied, the implications of these idioms for long-term code quality, particularly in ML projects—which predominantly employ often non-technical engineering expertise—remain underexplored. In ML projects, where complex algorithms, extensive data pipelines, and iterative model development coexist, the importance of maintaining clean, flexible code is magnified. This underscores the need to examine how Pythonic and non-Pythonic coding practices impact of code smells—indicators of poor design choices that degrade code quality over time [2].

In traditional software systems, code smells have been extensively studied as indicators of technical debt and reduced maintainability [3, 4, 5, 6]. However, the complexity of ML projects introduces unique challenges that make the study of code smell even more pressing. ML codebases are not limited to algorithmic logic—they also involve heavy data preprocessing, model training, tuning, and deployment [7]. As a result, they are particularly vulnerable to accumulating technical debt in the form of code smells, which can compromise the maintainability of ML projects [8].

This paper seeks to bridge the gap in the literature by conducting an empirical investigation into the role of Pythonic and non-Pythonic idioms in shaping code quality within ML projects. We explore how these idioms correlate with the presence of code smells, which serve as a proxy for maintainability issues in software [9]. The need for such a study is particularly relevant in the ML domain, where the velocity of code changes, experimental iterations, and the integration of cutting-edge algorithms amplify the risks associated with poor design choices.

Our study examines a large dataset of ML projects from the NICHE dataset [10], focusing on nine Pythonic and non-Pythonic idioms, and their relationship with the occurrence of code smells. We aim to uncover whether adherence to Pythonic

practices leads to cleaner, more maintainable code, particularly in reducing code smells that threaten the long-term viability of ML systems. Our statistical models highlight the prevalence of these idioms and examine their association with smells.

The results of our study reveal that "Truth Value Test" and "Assign Multi Targets" are the most common idioms in Pythonic and non-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 two main contributions: (a) a large-scale empirical study on the relation between the adoption of (non-)Pythonic idioms and code smells; (b) a publicly available replication package [11] that contains both scripts and data used for our analysis that other researchers can use to replicate and extend this work.

Structure of the paper. Section 2 summarizes the state-of-the-art and discusses the most closely related work. Section 3 provides an overview of the research method applied. Section 4 summarizes the results obtained. Section 5 provides additional investigation on Pythonic idioms and their impact on code quality. Section 6 discusses the implications of this work for the SE community. Section 7 discusses the potential threats to validity and the mitigation strategies applied and lastly, Section 8 concludes the paper.

2. Background and Related Work

This section provides the necessary information to understand the rest of the paper and summarize 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, emphasizing readability, simplicity, and conciseness. These principles are encapsulated in Python's unofficial mantra, *"The Zen of Python"* [1], 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.

Non-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. Refactoring non-Pythonic code into Pythonic code often involves replacing cumbersome and verbose constructs with more streamlined and effective idioms that Python provides.

For example, a common non-Pythonic approach might involve manually accumulating sums in a loop:

Listing 1: Non-Pythonic sum example

```
total = 0
for num in numbers:
    total += num
```

A Pythonic refactor of the above code would use the built-in `sum()` function, which is more concise and transparent:

Listing 2: Pythonic sum example

```
total = sum(numbers)
```

In the context of ML projects, writing Pythonic code might be particularly impactful. ML projects often involve complex data manipulations and transformations, areas where Pythonic idioms can significantly reduce code complexity and enhance maintainability. Moreover, these idioms can significantly boost the overall efficiency of the development process, a critical factor in performance-intensive applications such as ML projects. For instance, list comprehensions offer a more readable and efficient way to transform lists compared to traditional loops:

Listing 3: List comprehension example

```
squared = [x**2 for x in numbers]
```

This Pythonic idiom not only compacts the code but also enhances execution efficiency by optimizing the loop execution internally, which is crucial for data-intensive tasks in ML. Additionally, it aligns closely with the mathematical notation commonly used in ML algorithms, thus improving readability and maintainability of the code [12].

Moreover, Python's `with` statement, used for resource management, ensures that resources like file streams or sessions are properly managed without requiring explicit cleanup code, reducing the potential for errors and improving the robustness of data handling routines:

Listing 4: Using the with statement

```
with open('data.csv', 'r') as file:
    data = file.read()
```

In other terms, Pythonic code promises not just aesthetic and stylistic benefits but also practical advantages that may affect multiple properties of ML projects [12]. The Python's dominance in the ML domain, along with the potential benefits of Pythonic code, motivate our work: we aim at *empirically validating these claimed benefits and provide a data-driven understanding of how Pythonic coding influences ML projects*.

In doing so, we target the analysis of code smells, which in our study represent the proxy metric used to assess the impact of Pythonic code. Code smells are pivotal for assessing the impact of Pythonic coding practices on ML projects due to their indicative value concerning software quality and maintainability. As Pythonic code promotes readability, simplicity, and efficiency—core qualities that mitigate code complexity and enhance maintainability—examining code smells can provide empirical evidence on the effectiveness of these practices. Analyzing the prevalence and impact of code smells in Pythonic

versus non-Pythonic code helps validate the theoretical advantages of Pythonic coding, demonstrating its practical benefits in improving the operational efficiency and maintainability of ML systems.

2.2. Python-specific and ML-Specific Code Smells

Table 1: 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 any 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.

In the context of this study, we can distinguish two categories of smells: 1) *Python-specific* and *ML-specific code smells*. The first category includes smells that typically emerge during the development of traditional Python code. These smells are commonly related to software engineering issues, such as long methods, duplicated code, or long statements [13]. The second ones referred to potential design issues that can arise during the ML pipeline [14]. This category includes smells regarding wrong use of specific libraries used to facilitate the creation of ML pipelines (*e.g.,* Pandas or Pytorch) or best practice violations during the model creation. These smells can affect the ML model in multiple ways *e.g.,* altering the model performance or increasing the defect proneness.

In recent years, several smell detectors have been developed, most of which focus on identifying Python-specific code smells. However, many existing tools are outdated or no longer actively maintained (*e.g.,* PySmell) [13]. Among the more recent contributions, in 2025, Boloori et al. introduced DPy [13], a static analyzer designed to detect Python-specific smells. DPy can identify 11 such smells with recall and precision rates of 93% and 96%, respectively, demonstrating strong detection performance. Table 1 summarizes the smells detectable by DPy.

In the context of *ML-specific code smells*, Recupito *et al.* introduced CodeSmile in 2025 [15], a static analysis tool that processes Python source code by constructing an Abstract Syntax Tree (AST) to identify anti-patterns based on the catalog of ML-specific code smells defined by Zhang *et al.* [14]. The tool is capable of detecting 12 distinct ML-specific code smells, achieving recall and precision rates of 87% and 78%, respectively.

These smells reflect recurring issues that can negatively impact the quality, maintainability, or correctness of machine learning code, particularly in data preparation, model training, and evaluation workflows.

Table 2: List of ML-CSs detectable by CodeSmile.

Code Smell	Description	Pipeline Stage	Effect	Type
Chain Indexing (CIDX)	Occurs when a developer accesses a single element in a DataFrame using chained indexing with "[][]", which may degrade performance.	Data Cleaning	Performance	API-Specific
Columns and DataType Not Explicitly Set (CDE)	Happens when a DataFrame is created without clearly specifying column names and data types, leading to ambiguity.	Data Cleaning	Defect Proneness	Generic
Dataframe Conversion API Misused (DCA)	Arises when the ".values()" method is used to convert a DataFrame to a NumPy array, which can result in unintended consequences.	Data Cleaning	Defect Proneness	API-Specific
In-Place APIs Misused (IPA)	Occurs when a developer assumes a Pandas operation modifies data in-place, without using the "inplace=True" argument.	Data Cleaning	Defect Proneness	Generic
Gradients Not Cleared Before Backward Propagation (GNC)	Happens when "optimizer.zero_grad()" is omitted before "loss_fn.backward()", causing gradients to accumulate.	Model Training	Defect Proneness	API-Specific
Matrix Multiplication API Misused (MMA)	Occurs when "np.dot" is improperly used for matrix multiplication with NumPy arrays, reducing code clarity.	Data Cleaning	Readability	API-Specific
Memory Not Freed (MNF)	Happens when a model is repeatedly created in a loop without releasing memory via the appropriate library method.	Model Training	Memory Issue	Generic
Merge API Parameter Not Explicitly Set (MAP)	Arises when key parameters such as "how" and "on" are omitted in a Pandas merge, making the logic unclear.	Data Cleaning	Readability	Generic
NaN Equivalence Comparison Misused (NAN)	Occurs when comparisons to NaN are made using equality operators instead of "np.isnan()", which causes logic errors.	Data Cleaning	Defect Proneness	Generic
Pytorch Call Method Misused (PC)	Happens when a model's forward pass is executed using "self.net.forward()" rather than the preferred "self.net()".	Model Training	Robustness	API-Specific
TensorArray Not Used (TA)	Arises when "tf.constant" is used inside a loop to create arrays, instead of the more appropriate "tf.TensorArray()", risking inefficiency and errors.	Model Training	Efficiency & Defect Proneness	API-Specific
Unnecessary Iteration (UI)	Happens when loops are used instead of vectorized Pandas operations, resulting in reduced efficiency.	Data Cleaning	Efficiency	Generic

Table 2 provides an overview of the ML-specific code smells that CodeSmile is capable of detecting. As a concrete example, consider the *Gradients Not Cleared Before Backward Propagation* (GNC) smell. This issue occurs in PyTorch-based models when `optimizer.zero_grad()` is not called before `loss_fn.backward()` inside the training loop. Omitting this step causes gradients to accumulate across iterations, potentially leading to gradient explosion and an increased risk of defects.

2.3. Related Work

Fowler and Beck [2] 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 conventional software systems, e.g., systems developed in Java, investigating their origins, persistence, and mitigation strategies [16, 17, 18]. For instance, Tufano *et al.* [19] provided insights into the lifecycle of code smells, noting they are often introduced during initial file creation and usually removed when files are deleted.

Further studies have linked code smells to reusability mechanisms, such as inheritance and delegation, noting both their benefits and drawbacks [18]. Giordano *et al.* [20] found that while certain design patterns can reduce code smells, others may inadvertently foster them. Although the majority of this research has concentrated on Java systems [21, 22, 23, 5], studies by Vavrová *et al.* [24] 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.* [25] 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.* [26] 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.* [27] 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.

When comparing our work to those discussed above, we differentiate ourselves by specifically 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 leading to recommendations for ML practitioners who rely heavily on Python for implementing complex algorithms and data processing tasks.

On another note, our research contributes to advancing the current body of knowledge on Pythonic code and its practical impact. In this respect, Alexandru *et al.* [28] 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"* [1]. In contrast, Zid et al. [29] 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. Additionally, Leelaprute *et al.* [30] provided evidence that Pythonic idioms could significantly enhance both memory usage and execution times, suggesting areas for further practical research. Sakulniwat *et al.* [31] 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 that, in a non-negligible number of cases, developers improve their source code refactoring using Pythonic idioms.

Phan-udom *et al.* [32] released Teddy, a tool that provides Pythonic idioms examples starting from non-Pythonic idioms. The main limitation of this tool is that it does not perform refactoring operations; it only provides examples starting from a predefined knowledge base. Zhang *et al.* [33] introduced a tool for refactoring non-Pythonic code into Pythonic one, demonstrating its efficacy in real-world applications.

3. Research Method

The main *goal* of this study is to analyze the adoption of (non-)Pythonic idioms in ML projects and understand whether there is a relationship between idioms and the quality of the code, measured in terms of code smells within systems.

The *quality focus* of this study is to assess how the use of (non-)Pythonic idioms is related to the code smells arising.

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 aims to deepen their knowledge of both the adoption of Pythonic idioms in ML projects and the relationship between idioms and code smells. The *context* of our analysis is the NICHE [10] dataset, *i.e.,* a dataset containing 572 ML projects grouped into two categories "well-engineered" and "not-engineered"—according to eight dimension metrics.

Based on our motivations and according to our research objective, we asked:

> **🔍 RQ$_1$. On the Adoption of (non-) Pythonic Idioms.**
>
> *To what extent have (non-)Pythonic idioms been adopted in ML systems?*

With our **RQ**$_1$, we are interested in understanding the diffusion of both Pythonic and non-Pythonic idioms.

We split the **RQ**$_1$ into three sub-research questions.

More specifically, we asked:

- RQ1$_1$ What are the most frequent (non-)Pythonic idioms in ML projects?
- RQ1$_2$ How does the adoption of (non-)Pythonic idioms vary as the project grows in size?
- RQ1$_3$ What is the density of (non-)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 non-Pythonic idioms and the presence of code smells in ML systems?*

The main objective of the **RQ**$_2$ is to analyze to what extent (non-)Pythonic idioms are related to code smells from a statistical perspective.

Our investigation has a statistical connotation, *i.e.,* we addressed our research questions using statistical tests and models. We adopted the guidelines of Wohlin *et al.* [34] in terms of reporting and the *ACM/SIGSOFT Empirical Standards*;[1] in particular, we leveraged the "General Standard", "Data Science"

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

and "Repository Mining" guidelines. 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.,* non-Pythonic idioms) and built and tested a home-made tool to detect Pythonic snippets in repositories based on AST. After obtaining Pythonic idioms and non-Pythonic idioms, we integrated the results and calculated the frequencies, the variation, and the density in order to respond to our first research question. Finally, we ran SonarCloud [35] to extract information regarding project metrics *i.e.,* lines of code (LOC), File complexity, and number of commits, and number of bugs, DPy [13] to extract Python-specific code smells, and CodeSmile [15] to extract ML-specific code smells [14] from repositories, and applied statistical tests to analyze the relationship between Pythonic and non-Pythonic idioms with the rise of smells.

Figure 1 overview of the research process of this study.

3.1. Dataset Description and Project Selection

To conduct our investigation, we selected the NICHE dataset [10], which comprises 572 ML projects. Three main motivations drove our decision: 1) The dataset includes only popular, active, and with a substantial commit history projects *i.e.,* with at least 100 GitHub stars, a minimum of 100 commits, and a most recent commit dated after 1st May, 2020. These criteria help us exclude personal or inactive repositories. 2) The projects have been manually labeled by the dataset authors as either well-engineered or not-engineered, based on eight-dimensional metrics. This classification enables a comparative statistical analysis between the two groups, consisting of 441 well-engineered and 131 not-well-engineered projects. Table 3 outlines the eight metrics used to define well-engineered projects. A project is considered well-engineered if it satisfies at least four of the eight criteria. 3) The dataset encompasses a diverse range of project scopes, enhancing the generalizability of our findings.

Table 3: 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 shows sufficient documentation to permit 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 maturity of the community.
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.

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 (this allowed us to consider only projects mainly written in Python) and discarded projects not yet accessible on GitHub (*e.g.,* projects migrated to other system versioning). As a result of these steps, 303 projects were considered, of which 76 were "not engineered" and 227 were "well-engineered". Table 4 provides the descriptive statistics for the variables "Stars", "Commits", "NLOC", and "Pythonic Percentage" divided according to the column "Engineered".

Table 4: Descriptive Statistics for Variables "Stars", "Commits", "NLOC", and "Pythonic 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%
Not 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. RQ1: On the Idioms Diffusion

Table 5: List of Idioms Refactorizable 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 defining both the keys and values dynamically.
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 0 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.

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 [36], a tool designed to identify and refactor non-Pythonic idioms into Pythonic ones by analyzing the Abstract Syntax Tree (AST). The tool was evaluated on over 7,000 Python projects and achieved 100% of accuracy for all nine idioms. Table 5 describes the idioms refactorable by the tool. We instrumented RIdiom in order to count the instances of non-Pythonic idioms and refactor them into Pythonic equivalents. On the other hand, we needed to extract and count the Pythonic instances in software projects. To do this, we developed a tool that takes a Python project in input, analyzes all Python files, builds the corresponding AST, and verifies the presence of nodes and conditions that represent Pythonic idioms. To identify nodes and conditions, we applied similar rules used by RId-

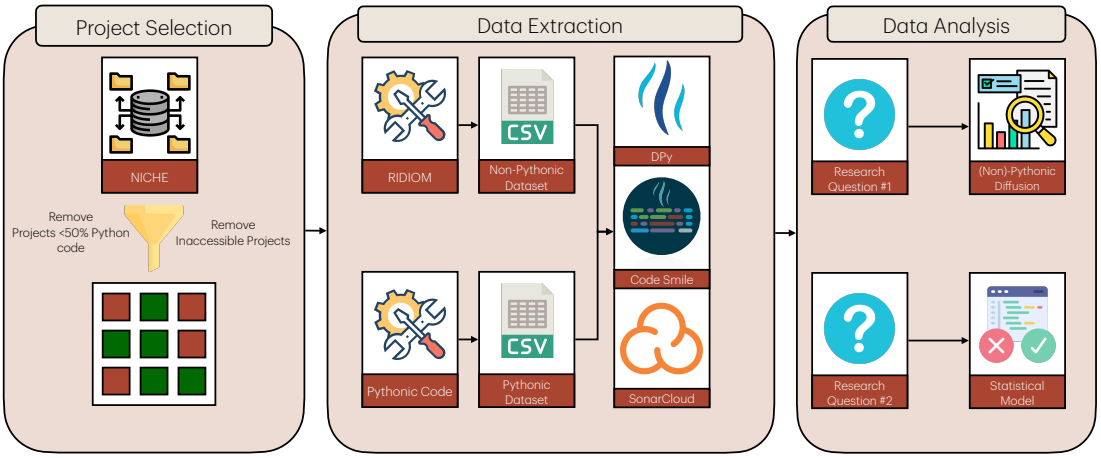

Figure 1: Research Method Overview

iom and also described in the "The Zen of Python" book [1]. The idioms that can be detected are outlined in Table 6. To enable more accurate comparisons, we implemented the detection only for idioms detectable by RIdiom.

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 Multi Targets	`ast.For`	`isinstance(node.target, ast.List)`
Loop Else	`ast.For`	`node.orelse Presence`

To address the $\mathbf{RQ1}_2$ and $\mathbf{RQ1}_3$, we calculated both the percentage of Pythonic idioms and the Pythonic density in the source code as follows:

$$\text{Pythonic Perc.} = \frac{\#PythonicIdioms}{\#PythonicIdioms + \#non - PythonicIdioms} \quad (1)$$

Equation 1 calculates the Pythonic Percentage, which measures the share of idiomatic (Pythonic) code patterns relative to all idioms (both Pythonic and non-Pythonic) in the codebase.

A higher value indicates a more idiomatic use of Python.

Finally, we calculated the density as follows:

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

Equation 2 computes the (non-)Pythonic Density, representing how often idioms (either Pythonic or non-Pythonic) appear per line of code (LOC). This helps assess how densely idiomatic patterns are used throughout the project.

3.3. RQ2: On the Relationship between Idioms and Smells

To answer $\mathbf{RQ}_2$, 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* [37], 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. Since the variables did not satisfy the normality assumption, we selected the Spearman test [38]—the non-parametric counterpart of Pearson [39]. We also normalize data by dividing per LOC. Furthermore, to mitigate the risk of false positives arising from multiple comparisons, we applied the Bonferroni correction to the resulting p-values [40].

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 (non)-Pythonic idioms and code smells, we considered code smells detectable by DPy and CodeSmile as dependent variables. In particular, we considered Python-specific code smells: *Long Statement*, *Long Parameter List*, *Long Method*, *Long Identifier*, *empty Catch Block*, *Complex Method*, *Complex Conditional*, *Missing Default*, *Long Lambda Function*, *Long Message Chain*, and *Magic number*.

While, as for ML-specific code smells, we considered: *Chain Indexing*, *Columns and DataType Not Explicitly Set*, *Dataframe Conversion API Misused*, *In-Place APIs Misused*, *Gradients Not Cleared Before Backward Propagation*, *Matrix Multiplication API Misused*, *Memory Not Freed*, *Merge API Parameter Not Explicitly Set*, *NaN Equivalence Comparison Misused*, *Pytorch Call Method Misused*, *TensorArray Not Used*, and *Unnecessary Iteration*.

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

Control Variables: As 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 [2]. 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.

All metrics were extracted as part of SonarCloud's output.

It is important to note two facts: On the one hand, we performed the same analysis considering Pythonic and non-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

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

Group	Smell Type	Mean	Std. Dev.	Median	Min	Max
Well Engineered	ML-Specific	7.07	14.94	1	0	90
	Python-specific	785.790	587,650	616	9	2,782
Not Well-Engineered	ML-Specific	2.64	4.81	0	0	22
	Python-specific	459,270	499,090	284	5	2,576

Before analyzing our results, we will provide a preliminary statistical description of the NICHE dataset of code smell diffusion.

Table 7 presents descriptive statistics on machine learning–specific and Python-specific code smells, comparing well-engineered and not well-engineered projects. For well-engineered projects, the average number of ML-specific smells is 7.07, whereas for Python-specific smells, it is significantly higher, with a mean of 785.790. In contrast, not well-engineered projects exhibit fewer ML-specific smells (mean = 2.64) and also fewer Python-specific smells on average (459.270). The standard deviation values indicate a high variability across projects, especially for Python-specific smells in both groups. These results suggest that well-engineered projects tend to contain more code (or more thorough detection), which is reflected in higher raw smell counts.

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, for each smell distribution, we normalized for KLOC, then using the Shapiro-Wilk [41] we verified normality. Since the distributions do not respect the normality assumption, we selected the Mann-Whitney U test [42], a non-parametric version of the T-Test [43].

Table 8 shows the results of the Mann-Whitney U test.

Table 8: Comparison of normalized code smells (per 1K LOC) with significance and effect size.

Smell Type	Well-engineered	Not well-engineered	P-value	Effect Size
Python-specific	0.93	0.66	0.295	0.55
ML-specific	103.20	112.14	0.477	0.976

As we can observe from the table, in all cases, the p-value is greater than 0.05, indicating no statistically significant differences in the distribution of smells between Python-specific code smells and ML-specific ones. However, even though the p-value does not indicate statistical significance, the effect size, particularly for ML-specific smells, suggests that the observed differences may be practically meaningful. This could imply that well-engineered practices could substantially influence the presence and distribution of code smells, especially within ML systems.

4.1. $RQ1_1$: On the Frequencies of Idioms in ML Projects

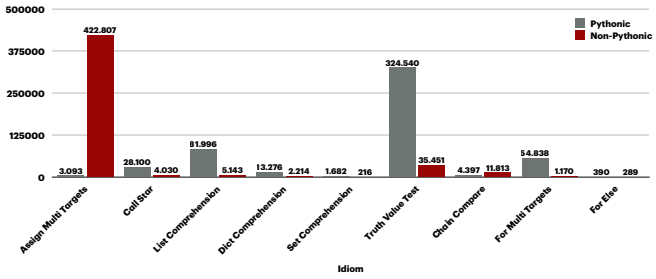

Figure 2: Frequencies of Pythonic and Non-Pythonic Idioms

Figure 2 displays the frequencies of Pythonic (gray) and non-Pythonic (red) idioms. Among Pythonic idioms, the most frequent is the *Truth Value Test*, followed by *List Comprehension* and *For Multi Targets*. These idioms enable concise and readable code, particularly when dealing with loops or complex data manipulations. In addition, we can note that among the "comprehension" constructs, "*List Comprehension*" is notably the most widely used, far surpassing other forms like dictionaries or set comprehensions.

Moving on to non-Pythonic idioms, the most frequent idiom observed is "*Assign Multi Targets*" followed by "*Truth Value Test*" and "*Chain Compare*". These results indicate that practitioners may frequently write multiple assignment statements separately rather than utilizing Python's ability to assign multiple variables in a single statement.

It is important to remark that in both Pythonic and non-Pythonic idioms, the less frequent idiom is "*For Else*", suggesting the unpopularity of this construct in ML projects.

Results $RQ1_1$

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

4.2. $RQ1_2$: On the Variation of Project Size

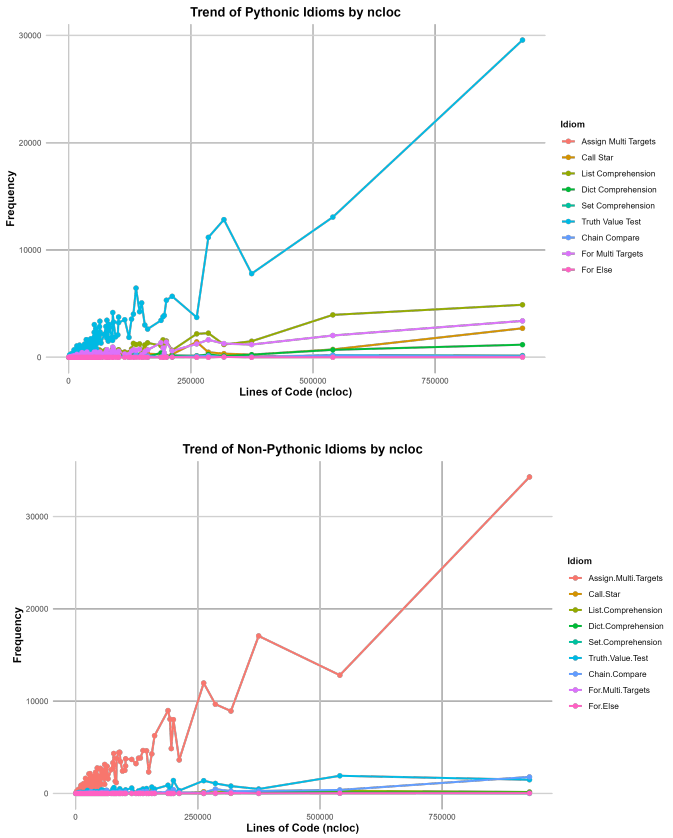

Figure 3: Variation of Pythonic Idioms and Non-Pythonic Idioms.

Figure 3 illustrates the variation in the usage of Pythonic idioms (upside) and non-Pythonic idioms (downside) with respect to the increase in NLOC. As shown in the figure, the most frequently adopted Pythonic idiom is the "*Truth Value Test*", whereas the most prevalent non-Pythonic, refactorable idiom is "*Assign Multi Targets*". In both cases, it is possible to observe that the growth of the "*Truth Value Test*" and "*Assign Multi Targets*" idioms accelerates dramatically after 250,000 lines. This trend suggests that as the project size increases, developers are more inclined to adopt both Pythonic and non-Pythonic idioms. 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 non-idiomatic patterns that are prevalent in larger projects, underscoring the need for targeted refactoring efforts.

Results $RQ1_2$

The results of $RQ1_2$ shows that the most prevalent Pythonic is "*Truth Value Test*" and the most prevalent non-Pythonic idiom is *Assign Multi Targets*.

4.3. $RQ1_3$: On the Density of Idioms

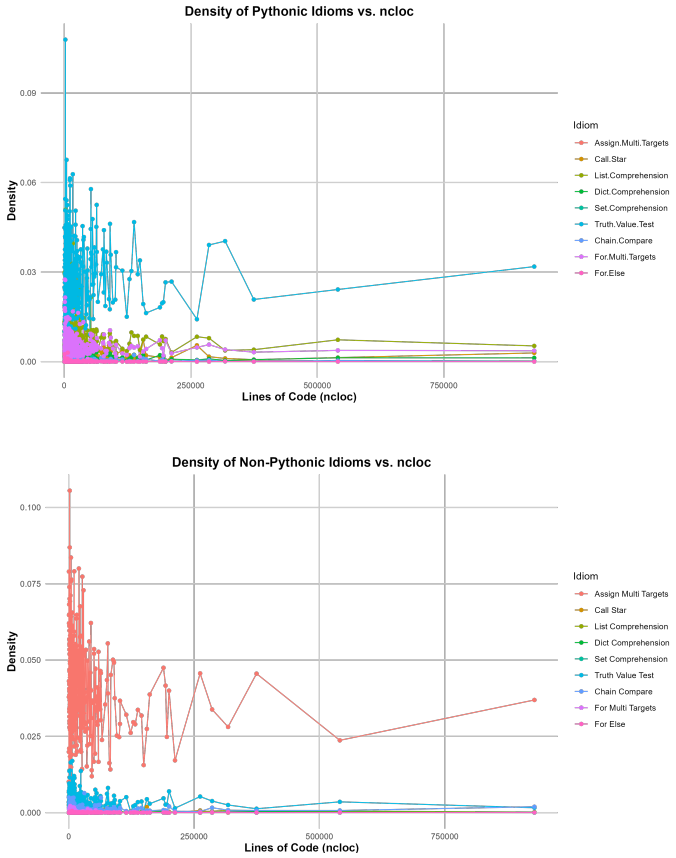

Figure 4: Density of Pythonic and Non-Pythonic Idioms

Figure 4 shows the density of Pythonic idioms (upside) and non-Pythonic idioms (downside). As shown from the figure, still in this case, the most adopted Pythonic idiom in a term of density is "*Truth Value Test*", and the most density non-Pythonic idiom is "*Assign Multi Targets*". These results reinforce the outcome of the $RQ1_2$, suggesting that these idioms are largely adopted in software systems.

Results $RQ1_3$

The results of $RQ1_3$ indicate that the most adopted pythonic idiom in terms of density is "*Truth Value Test*", while the most re-factorizable idiom is "*Assign Multi Targets*".

4.4. RQ2: On the Relationship Between Idioms and Smells

Table 9 shows the results of our statistical test after the application of the Bonferroni correction.

Even after normalizing, certain non-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 *Magic Number*, *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 *LOC*—dominate: $\rho(\text{LOC}, \text{Long Statement}) = 0.73$ and $\rho(\text{LOC}, \text{Long Method}) = 0.69$, indicating that file size is the strongest predictor of code smell incidence. *File Complexity* correlates with *Complex Method* at $\rho = 0.37$ ($p < 10^{-11}$, ***), and *Commits* also shows moderate correlations with several smells.

Table 9: Spearman Correlation between (Non-)Pythonic Idioms and Code Smells (normalized by NLOC), with Bonferroni-corrected significance levels

Idiom	Smell	Spearman$_\rho$	p-value	Significance
Non-Pythonic Call Star	Magic Number	0.36	7.19e-11	***
Non-Pythonic Chain Compare	Complex Conditional	0.35	2.92e-10	***
Non-Pythonic Call Star	Long Parameter List	0.34	1.86e-09	***
Assign Multi Targets	Long Parameter List	0.30	7.30e-08	***
Non-Pythonic Chain Compare	Complex Method	0.30	1.23e-07	***
Non-Pythonic Call Star	Complex Method	0.30	1.35e-07	***
Non-Pythonic Call Star	Long Method	0.28	9.90e-07	***
Non-Pythonic Call Star	Complex Conditional	0.26	3.92e-06	**
Non-Pythonic List Comprehension	Complex Method	0.26	4.79e-06	**
Non-Pythonic Truth Value Test	Gradients Not Cleared Before Backward Propagation	0.25	1.05e-05	**
Truth Value Test	Complex Method	0.25	1.17e-05	**
Non-Pythonic Chain Compare	Long Parameter List	0.25	1.47e-05	**
Set Comprehension	Empty Catch Block	0.24	2.50e-05	**
Non-Pythonic Chain Compare	Magic Number	0.24	2.68e-05	**
Non-Pythonic Chain Compare	Long Method	0.24	2.73e-05	**
Assign Multi Targets	Magic Number	0.23	4.71e-05	*
Assign Multi Targets	Complex Method	0.23	5.20e-05	*
Non-Pythonic Call Star	Long Statement	0.23	5.66e-05	*
Truth Value Test	Complex Conditional	0.23	5.85e-05	*
Truth Value Test	Gradients Not Cleared Before Backward Propagation	0.23	6.25e-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	*
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	Magic Number	0.62	4.94e-34	***
NLOC	Long Parameter List	0.61	1.16e-31	***
NLOC	Long Identifier	0.50	8.30e-21	***
NLOC	Complex Conditional	0.41	7.72e-14	***
NLOC	Long Lambda Function	0.31	2.31e-08	***
NLOC	Columns And Datatype Not Explicitly Set	0.29	3.00e-07	***
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 Identifier	0.24	1.80e-05	*
Commits	Long Method	0.22	8.43e-05	*

* $p < 0.1$; ** $p < 0.05$; *** $p < 0.01$

Restricting attention to *well-engineered* projects (Table 10), we observe that the same idioms (*Call Star*, *Chain Compare*) still correlate moderately with Python-specific smells ($\rho \approx 0.30$–0.36), indicating that even in high-quality codebases, these idioms cooccur with issues like *Magic Number* and *Complex Conditional*. However, control variables remain dominant: $\rho(\text{Loc}, \text{Long Statement}) = 0.71$ and $\rho(\text{Loc}, \text{Long Method}) = 0.69$, underscoring that file size and complexity are the primary determinants of the incidence of smell, even when engineering practices are rigorous.

In *not-well-engineered* projects (Table 11), control variables alone dominate the top correlations: $\rho(\text{Loc}, \text{Long Statement}) = 0.71$, $\rho(\text{Loc}, \text{Long Method}) = 0.67$, $\rho(\text{Loc}, \text{Long Parameter List}) = 0.64$, and $\rho(\text{Loc}, \text{Magic Number}) \approx 0.63$. *File Complexity* remains strongly linked to *Complex Method* ($\rho = 0.48$, $**$). No idiom-based correlations appear here, suggesting that in poorly governed codebases, raw size and complexity fully eclipse idiomatic style as predictors of code smells.

Results RQ2

Considering all projects—well- and not-well-engineered—we found strong correlations between some non-Pythonic idioms (*e.g.,* Call Star, Chain Compare) and Python-specific code smells (*e.g.,* Magic Number, Complex Condition). Some Pythonic idioms (*e.g.,* Assign Multi Targets) also strongly correlate with Python-specific smells (*e.g.,* Long Parameter List). Results suggest a weak correlation between idioms and ML-specific smells. In well-engineered projects, Pythonic idioms correlate with Python-specific smells (*e.g.,* Call Star with Long Parameter List), while in not-well-engineered ones, only control variables relate to some smells.

Table 10: Spearman Correlation between (non-)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	***
Call Star	Magic Number	0.32	7.09e-07	***
Chain Compare	Complex Method	0.31	1.78e-06	***
Call Star	Complex Method	0.29	6.26e-06	**
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	*
Truth Value Test	Empty Catch Block	0.26	5.36e-05	*
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	Magic Number	0.63	1.02e-09	***
NLOC	Long Parameter List	0.61	5.39e-08	***
NLOC	Long Identifier	0.46	2.19e-06	***
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	*
Commits	Empty Catch Block	0.27	4.75e-05	*

* $p < 0.1$; ** $p < 0.05$; *** $p < 0.01$

Table 11: Spearman Correlation (non-)Pythonic Idioms and Code Smells for Not-Well-Engineered Projects (normalized by NLOC), with Bonferroni-corrected significance levels

Idiom	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	Magic Number	0.63	1.02e-09	***
NLOC	Complex Method	0.63	8.29e-10	***
File Complexity	Complex Method	0.48	1.19e-05	**

* $p < 0.1$; ** $p < 0.05$; *** $p < 0.01$

5. Further Analysis

Building on previous findings that identified correlations between Pythonic idioms and code smells, we sought to further investigate the role of idiomatic usage in broader software quality concerns. Given that code smells are often considered early indicators of defects, it is reasonable to ask whether idioms associated with smells may also be linked to actual software bugs. This follow-up analysis explores that possibility, aiming to uncover whether idiomatic patterns correlate with the presence of bugs in Python code.

Using SonarCloud, we extracted the number of reported bugs across 303 projects. We then applied the Spearman correlation test [38] to examine the statistical relationship between the number of bugs and the usage frequency of nine (non-)Pythonic idioms. These idioms served as independent variables, with the bug count as the dependent variable. As control variables, we included the same three metrics used in **RQ2** *i.e.,* NLOC, complexity, and number of commits.

To ensure consistency, we maintained the project categorization based on the *Engineered* column, separating the analysis into well-engineered and not-well-engineered projects.

Table 12 presents the Spearman correlation results for well-engineered projects. As shown in the table, most (non-)Pythonic idioms exhibit statistically significant positive correlations with bug counts in well-engineered projects. non-Pythonic idioms such as *non-Pythonic Assign Multi Targets*, *non-Pythonic Truth Value Test*, and

Table 12: Spearman Correlation Between (non)-Pythonic Idioms and Bugs for Well-Engineered Projects.

Idiom	Spearman$_\rho$	p-value	Significance
Non-Pythonic Assign Multi Targets	0.64	1.18e-27	***
Non-Pythonic Call Star	0.30	4.37e-06	***
Non-Pythonic List Comprehension	0.42	6.02e-11	***
Non-Pythonic Dict Comprehension	0.32	9.89e-07	***
Non-Pythonic Set Comprehension	0.12	7.42e-02	*
Non-Pythonic Truth Value Test	0.58	9.53e-22	***
Non-Pythonic Chain Compare	0.58	1.71e-21	***
Non-Pythonic For Multi Targets	0.17	8.48e-03	***
Non-Pythonic For Else	0.34	1.29e-07	***
Assign Multi Targets	0.37	7.54e-09	***
Call Star	0.50	7.21e-16	***
List Comprehension	0.54	1.35e-18	***
Dict Comprehension	0.39	1.42e-09	***
Set Comprehension	0.27	2.91e-05	***
Truth Value Test	0.59	1.82e-22	***
Chain Compare	0.38	2.88e-09	***
For Multi Targets	0.50	6.38e-16	***
For Else	0.28	1.41e-05	***
Commits	0.14	3.13e-02	**
complexity	0.62	2.86e-25	***
ncloc	0.63	6.10e-27	***
* $p < 0.1$; ** $p < 0.05$; *** $p < 0.01$			

Table 13: Spearman Correlation Between (non)-Pythonic Idioms and Bugs for Not Well-Engineered Code.

Idiom	Spearman$_\rho$	p-value	Significance
Non-Pythonic Assign Multi Targets	0.56	1.67e-07	***
Non-Pythonic Call Star	0.50	4.29e-06	***
Non-Pythonic List Comprehension	0.37	9.72e-04	***
Non-Pythonic Dict Comprehension	0.27	1.92e-02	**
Non-Pythonic Truth Value Test	0.52	1.73e-06	***
Non-Pythonic Chain Compare	0.44	6.63e-05	***
Non-Pythonic For Multi Targets	0.42	1.66e-04	***
Non-Pythonic For Else	0.29	1.02e-02	**
Assign Multi Targets	0.45	4.04e-05	***
Call Star	0.38	6.96e-04	***
List Comprehension	0.46	3.21e-05	***
Dict Comprehension	0.29	1.23e-02	**
Truth Value Test	0.53	8.12e-07	***
Chain Compare	0.36	1.27e-03	***
For Multi Targets	0.44	6.30e-05	***
For Else	0.29	1.22e-02	**
Commits	0.22	5.33e-02	*
complexity	0.56	1.55e-07	***
ncloc	0.61	5.09e-09	***
* $p < 0.1$; ** $p < 0.05$; *** $p < 0.01$			

non-Pythonic Chain Compare demonstrate strong correlations, suggesting that they are potentially misuse-prone forms may be linked to higher fault-proneness.

Even Pythonic idioms such as *List Comprehension*, *Truth Value Test*, and *For Multi Targets*—show moderate to strong positive correlations with bugs. This may indicate that, despite their nature, excessive or inappropriate usage of these constructs can introduce complexity and increase the likelihood of defects.

The control variables—complexity, NLOC, and number of commits—also display significant positive correlations with bug counts, aligning with established findings in software quality literature.

Table 13 summarizes the results for not-well-engineered projects.

Unlike in **RQ2**, where correlations for not-well-engineered projects were mostly limited to control variables, this extended analysis reveals statistically significant correlations between many (non-)Pythonic idioms and bug counts. This suggests that in less well-structured codebases, Pythonic constructs—whether used properly or not—may still be indicative of increased fault potential, perhaps due to their misuse in less maintainable or complex contexts.

6. Discussion and Implications

Our findings offer several implications for both practitioners and researchers, which we explore in the following.

Does Pythonic really improve code quality? From **RQ1**, we observe that idioms such as "Truth Value Test" and "Assign Multi Targets" are among the most frequently used across ML projects—both in absolute frequency and density. However, **RQ2** 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", "Empty Catch Block", and the "Gradients Not Cleared Before Backward Propagation". Similarly, "Assign Multi Targets" shows moderate correlation with "Long Parameter List" and weak correlations with "Magic Number", "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. This applies not only to idioms present in the original code but also to those introduced via refactoring from non-Pythonic patterns. Moreover, we find that such correlations are not exclusive to Pythonic constructs: many non-Pythonic idioms refactorable into Pythonic style also show statistically significant associations with Python-specific smells, implying that the risk is not in the idioms themselves but in how and where they are used.

Thus, the effectiveness of Pythonic idioms appears to be *context dependent*. Their misuse or overuse can contribute to code smells in ways that contradict their intended purpose. Consequently, developers should avoid blind adherence to stylistic norms and instead assess whether idiomatic constructs truly improve clarity, modularity, and maintainability in their specific context. Similarly, refactoring tools and linters should avoid stylistic suggestions that ignore structural implications.

☞ *Applying idioms or refactoring to "Pythonic" style does not guarantee cleaner or more maintainable code; misapplication may introduce or obscure structural issues.*

Pythonic in ML Projects: Yes, No, or Nah? The role of Pythonic idioms in ML projects is less definitive when examined through the lens of **RQ2**. Although idioms like Truth Value Test appear in both Pythonic and refactorable code and are associated with traditional smells, our analysis finds no strong or consistent relationships between any idiom and ML-specific code smells. For instance, while some weak correlations exist (*e.g.,* between "Truth Value Test" and "Gradients Not Cleared Before Backward Propagation"), most ML-related issues appear largely unaffected by idiomatic style.

This disconnect points to an important insight: the quality challenges of ML codebases are not primarily idiomatic. ML-specific smells, such as improper use of framework APIs (*e.g.,* PyTorch or pandas), unsafe memory management, or ineffective data transformations, stem from domain-specific concerns rather than general-purpose code style. Therefore, writing Pythonic code in an ML pipeline may improve local clarity but does not mitigate issues like silent data corruption, model leakage, or performance bottlenecks caused by incorrect

API use.

From this perspective, Pythonic idioms in ML projects are stylistically useful but not quality-critical. Their use should be considered a secondary concern, subordinate to ensuring semantic correctness and safe API usage. For researchers, this distinction suggests that ML code quality requires different models and tools than traditional software. For practitioners, it implies that attention should shift from stylistic polishing to more robust handling of domain-specific practices.

☞ *Traditional software metrics and smells may not capture the core risks of ML development. New, ML-aware quality models are needed.*

7. Threats to Validity

This section summarizes the main threat to the validity of our study and discusses the mitigation strategies applied.

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 two already validated tools, *i.e.,* we selected DPy for Pythonic-specific code smells, and CodeSmile for ML-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 over 50% of the code was in Python. This approach helps avoid noise from non-Pythonic sources, such as code written in other languages. To identify non-idiomatic Python code, we used the RIdiom tool, the only validated tool in the literature capable of detecting code that could be refactored to become more Pythonic. 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. 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 RQ2, we examined the relationship between Pythonic idioms and code quality, measured via code smells. While collecting data through DPy and CodeSmile, 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.

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. For the t-test, we checked for normality, and when the data did not satisfy these conditions, we avoided models requiring normal distribution or homoscedasticity. We set a significance level of 0.05, which was consistently respected across all tests and models. While the study is quantitative, further qualitative research is needed to understand how Pythonic idioms impact code quality, as this study serves as a preliminary exploration.

External Validity. The primary threat to external validity stems from the dataset used. The selection of projects is crucial, so we utilized the NICHE dataset, a large collection of real-world ML-enabled systems, encompassing 303 projects. 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 non-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.

8. Conclusions

This paper proposed a large-scale empirical investigation into the relationship between Pythonic and non-Pythonic idioms and code smells in ML projects. We analyzed 303 Python projects classified as "well-engineered" or "not-engineered", discovering that Pythonic idioms more frequents are "Truth Value Test" and "Assign Multi Target" for "well-engineered" and "not-well-engineered", respectively.

Our statistical analysis revealed a correlation between the usage of (non-)Pythonic idioms and code smells, indicating that refactoring non-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 smell, especially in well-engineered projects. We finally provided implications and discussions to drive further research on the matter.

Acknowledgment

TDB

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 has data included 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.

Credits

References

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

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

[3] F. A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: An experimental assessment., J. Object Technol. 11 (2012) 5–1.

[4] F. Palomba, Textual analysis for code smell detection, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 2, IEEE, 2015, pp. 769–771.

[5] 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.

[6] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Lightweight detection of android-specific code smells: The adoctor project, in: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), IEEE, 2017.

[7] A. Paleyes, R.-G. Urma, N. D. Lawrence, Challenges in deploying machine learning: a survey of case studies, ACM computing surveys 55 (2022) 1–29.

[8] 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, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 420–430. URL: https://doi.org/10.1145/3379597.3387479. doi:10.1145/3379597.3387479.

[9] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, Code smells and refactoring: A tertiary systematic review of challenges and observations, Journal of Systems and Software 167 (2020).

[10] 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, ????

[11] Anonymous, On the adoption of pythonic idioms in machine learning projects and their effect on code smells, ???? URL: https://figshare.com/s/6aeca03a15631f306e5a.

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

[13] A. Boloori, T. Sharma, Dpy: Code smells detection tool for python (????).

[14] H. Zhang, L. Cruz, A. Van Deursen, Code smells for machine learning applications, in: Proceedings of the 1st international conference on AI engineering: software engineering for AI, 2022, pp. 217–228.

[15] G. Recupito, G. Giordano, F. Ferrucci, D. Di Nucci, F. Palomba, When code smells meet ml: on the lifecycle of ml-specific code smells in ml-enabled systems, arXiv preprint arXiv:2403.08311 (2024).

[16] 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 (????) 243–275.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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, ????

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

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

[38] 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.

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

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

[41] 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.

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

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