"You Have Said Too Much": Java-Like Verbosity Anti-patterns in Python Codebases

Yuzhi Ma
Eli Tilevich
{yuzhima,tilevich}@vt.edu
Software Innovations Lab, Virginia Tech
Blacksburg, VA, USA

Abstract

As a popular language for teaching introductory programming, Java can profoundly influence beginner programmers with its coding style and idioms. Despite its many advantages, the paradigmatic coding style in Java is often described as verbose. As a result, when writing code in more concise languages, such programmers tend to emulate the familiar Java coding idioms, thus neglecting to take advantage of the more succinct counterparts in those languages. As a result of such verbosity, not only the overall code quality suffers, but the verbose non-idiomatic patterns also render code hard to understand and maintain. In this paper, we study the incidences of Java-like verbosity as they occur in Python codebases. We present a collection of Java-Like Verbosity Anti-patterns and our pilot study of their presence in representative open-source Python codebases. We discuss our findings as a call for action to computing educators, particularly those who work with introductory students. We need novel pedagogical interventions that encourage budding programmers to write concise idiomatic code in any language.

CCS Concepts: • **Software and its engineering → Multiparadigm languages**.

Keywords: code quality, verbosity, anti-patterns, Java, Python, CS education

ACM Reference Format:
Yuzhi Ma and Eli Tilevich. 2021. "You Have Said Too Much": Java-Like Verbosity Anti-patterns in Python Codebases. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3484272.3484960

SPLASH-E '21, October 20, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9089-7/21/10...$15.00
https://doi.org/10.1145/3484272.3484960

1 Introduction

In a time of drastic change it is the learners who inherit the future. The learned usually find themselves equipped to live in a world that no longer exists.

Eric Hoffer

When learning how to write computer programs, learners are introduced to the fundamental computing concepts in a programming language that becomes their *first programming language*. Without any prior exposure to programming, introductory learners naturally see their first language's idioms and style as standard building blocks for solving programming problems [1, 2]. As computing learners are gaining experience and are introduced to other languages, they often try to emulate the familiar mechanisms of their first language, but by means of other languages, particularly if these other languages support the same paradigm (e.g., imperative or object-oriented). However, if a new language's common programming idioms and style differ from the programmer's first language, the resulting code quality suffers, with unnecessary verbosity becoming a common harmful side affect.

Due to the success of Java in both commercial and educational settings [3, 4], computing educators have long embraced this language for teaching introductory computing concepts. Numerous programmers all over the world have had Java as their first programming language. In recent years, Python has increasingly become the language of choice for various software development pursuits, particularly due to the rise of data science [8]. Although Python has also been an immensely popular language for teaching introductory programming, numerous Python programmers have been groomed into computing in Java. The vast similarities between Java and Python become a double-edged sword. On the one hand, Java programmers can learn Python easily and quickly start contributing working code to realistic projects. On the other hand, when writing code in Python, former Java programmers may be tempted to emulate the familiar Java idioms and style in Python, thus writing non-idiomatic code, which is likely to end up more verbose than its idiomatic Python counterparts. On average, Python tends to promote a programming style than is more concise than that of Java

[7]. That is, the same functionality can be implemented more concisely in Python than in Java.

The presence of non-idiomatic Python code hinders code quality, making the resulting codebases hard to understand. In addition, the amount of software maintenance effort required to address changed requirements or fix defects has been found to be proportional to the size of a program [6]. Hence, the presence of verbose code can inflate maintenance efforts and costs.

To address this problem, this paper studies the problem of Java-like verbosity in representative Python codebases. We define six Java-like verbosity anti-patterns by explaining their origins and how they should be refactored into their concise Python-native counterparts. We then present the results of a pilot study that analyzed representative open-source codebases for the presence of the aforementioned anti-patterns. We conclude the paper by presenting a call for action to computing educators, who can take simple but effective steps that would encourage their students to write concise idiomatic code in any language.

The rest of this paper is structured as follows. Section 2 presents a catalog of our anti-patterns. Section 3 presents the results of our pilot study. Section 4 discusses our findings and their implications. Section 5 gives an overview of related approaches, and Section 6 presents concluding remarks.

2 Java-Like Verbosity Anti-patterns

In this section, we present a catalog of six Java-like verbosity anti-patterns in Python. We name each anti-pattern and present its concise, idiomatic "Pythonic" counterpart. The pilot study in Section 3 identifies the incidences of some of these anti-patterns in representative open-source Python codebases.

2.1 Origin and Methodology

We came up with the idea to define these anti-patterns having examined a sampling of programming assignments, submitted by undergraduates in a Python course. This course was introduced with the specific goal to acquaint students with working knowledge of Python to prepare them for subsequent courses in Data Science. Most students in the course have successfully completed an introductory sequence of three programming courses, all of which were taught in Java.

Having carefully looked through students' code, we identified eleven anti-patterns, which manifested themselves prominently in student code. We further examined several popular open-source Python projects to determine whether the observed anti-patterns were present there as well. Only six out of the original eleven anti-patterns that were present both in student-written code and in open-source projects were selected as the anti-patterns described below. We would be amiss if we did not mention up front that we did not systematically examine how prevalent these anti-pattern candidates were in open-source codebases, leaving this task to be investigated as a future work direction.

2.2 Format

In describing our anti-patterns, we follow this format:

1. Name of this Anti-pattern
2. The programming scenarios under which this anti-pattern most commonly occurs
3. Possible origin (i.e., why Java programmers are prone to write code like that in Python)
4. Explanation of the concise counterpart

2.3 Anti-pattern #1: "Modifying a list with an unnecessary if statement"

```python
#Anti-pattern
list = [0,1,2,3,4,5,6,7,8,9,10]
odd_list = []
for x in list:
  if x % 2 == 1:
    odd_list.append(x)

#Concise counterpart
list = [0,1,2,3,4,5,6,7,8,9,10]
odd_list = [x for x in list if x % 2==1]
```

1. "Modifying a list with an unnecessary if statement"
2. This anti-pattern is introduced when filtering list values on simple conditions.
3. Java programmers are used to writing `for` loops to iterate through lists, filter out values with an `if`, and may be unfamiliar with the concise square brackets mechanism to combine a `for` loop and an `if` condition statement.
4. In Python, one can specify operations on lists more declaratively by listing the modification operation and an iterator (i.e., `if`) over a collection

2.4 Anti-pattern #2: "Unnecessary function definition"

```python
#Anti-pattern
def add(x,y):
  return x+y

#Concise counterpart
add = lambda x,y:x+y
```

1. "Unnecessary function defintion"
2. This anti-pattern is introduced whenever we need a function to do simple calculations multiple times.

3. Java programmers are used to define a helper function, and may be unfamiliar with the `lambda` construct in Python.
4. In Python, one can employ `lambda` to define concise helper functions on the fly.

2.5 Anti-pattern #3: "For loop instead of function"

```python
#Anti-pattern
list = ["a", "b","c","a","a"]
count = 0
for x in list:
  if x == "a":
    count += 1

#Concise counterpart
list = ["a", "b","c","a","a"]
count = list.count("a")
```

1. "For loop instead of function"
2. This anti-pattern is introduced whenever we need to operate on a collection, such as counting how many times a value occurs in a list.
3. Java programmers are used to writing `for` loops to iterate through lists and use `if` statements to count the occurrences, and may be unfamiliar with the concise built-in `count` function in Python.
4. In Python, one can employ `count` to count the number of occurrences of a value in a list.

2.6 Anti-pattern #4: "Over-complicated comparison"

```python
#Anti-pattern
result = 2
if(result <= 3 and result > 1):
    print(True)

#Concise counterpart
result = 2
print(1 < result <=3)
```

1. "Over-complicated comparison"
2. This anti-pattern is introduced whenever we compare a value with any other two values
3. Java does not support comparisons of a value with two others together, so Java programmers are used to comparing them individually.
4. In Python, one can compare a value with any other two values in a single statement.

2.7 Anti-pattern #5: "Over-complicated negative indexing"

```python
#Anti-pattern
a = [0,1,2,3,4,5,6,7,8,9,10]
last = a[a.length - 1]

#Concise counterpart
a = [0,1,2,3,4,5,6,7,8,9,10]
last = a[-1]
```

1. "Over-complicated negative indexing"
2. This anti-pattern is introduced whenever we need to access some elements at the end of a list
3. Java does not support negative indexing. Thus, Java programmers get used to employing `length` minus some numbers to get the needed element.
4. In Python, one can use negative indexing to access any last elements of a list directly.

2.8 Anti-pattern #6: "An unnecessary intermediate tuple for multi-returns"

```python
#Anti-pattern
def triple():
  return 0, 1, 2
result = triple()
zero = result[0]
one = result[1]
two = result[2]

#Concise counterpart
def triple():
  return 0, 1, 2
zero, one, two = triple()
```

1. "An unnecessary intermediate tuple for multi-returns"
2. This anti-pattern is introduced when assigning multiple returned values to multiple variables.
3. Java programmers are used to writing separate statements to assign a value to a variable.
4. In Python, one can assign multiple returned values to multiple variables in a single statement.

3 Locating Verbosity Anti-patterns in Open-Source Codebases

To understand how prevalent the anti-patterns discussed above are in open-source codebases, we conducted a pilot study. We selected four anti-patterns: "Unnecessary function definition", "An unnecessary intermediate tuple for multi-returns", "Over-complicated comparison", and "Modifying a list with an unnecessary if statement".

We employed Comby[1], an open-source tool hosted on GitHub. Comby automates the process of structural code searching. By using this tool, we used Comby's declarative rules to describe the structural patterns that express each target anti-pattern. For example, to find `if` statements, such as `if (a > b)`, Comby can be given the following rule:

[1]https://github.com/comby-tools/comby

Table 1. The results of a pilot study of the prevalence of anti-patterns in open-source codebases.

Empirical Analysis			
Project Name / Anti-pattern Name	TensorFlow Model Garden	Manim	Snips
# Project Size (LOC)	401,490	294,309	26,411
"Unnecessary function definition"	1138	989	162
"Over-complicated comparison"	34	7	1
"An unnecessary intermediate tuple for multi-returns"	10	3	1
"Modifying a list with an unnecessary if statement"	29	3	0

`if(:[condition])`. Given this pattern and a Python source file, Comby returns a list of line numbers, at which the matched source code patterns were detected.

Having parameterized Comby with this input, we then ran the tool on three Python open-source projects—TensorFlow Model Garden, Manim, and Snip—and tabulated the total found number of occurrences of each target anti-pattern. TensorFlow Model Garden is to provide the users of the popular TensorFlow machine learning framework with state-of-the-art (SOTA) models and modeling solutions. Manim is a framework that creates animations as explanatory math videos. Snip is a natural Language Processing library for parsing natural language sentences to extract their structured information. The total combined lines of code across all three subjects is above 700K.

The pilot study's results appear in Table 1. The "unnecessary function defintion" is by far the most prevalent anti-pattern. We searched for all named functions whose body comprises only one line of code. For programmers introduced to the discipline in a language without support for anonymous functions[2], using Python `lambda` would be unfamiliar, so such programmers tend to introduce named functions even for short code snippets.

"Over-complicated comparison" is the second most prevalent anti-patterns detected, although it is not as common as "unnecessary function definition." These features are possibly unique to Python, and unless introduced to them explicitly, programmers would be likely to use more verbose alternatives, directly translated from similar idioms in Java or other languages.

Threats to Validity. The validity of the results of our pilot study is threatened by our reliance on Comby, which may have misidentified some anti-patterns. In a follow-up study, it may be prudent to carefully verify the veracity of Comby's matching of each target anti-pattern. If the results prove unsatisfactory, a more elaborate search utility may need to be employed. Nevertheless, for a pilot study these results need not to be definitive, as our goal is to draw the attention of the research community to a new issue in code quality rather than to report confirmed final findings.

4 Discussion and Call for Action

Programmers should be encouraged to write concise idiomatic code in any language. If the same functionality can be expressed in fewer lines of code without losing readability, the resulting codebase becomes easier to understand and modify for other programmers. Concise idiomatic code also facilitates various performance optimizations. However, absolute conciseness is not always a legitimate objective, as the ultimate goal is to maximize readability. If a more concise version would be harder to read, then the more verbose version should be retained. For example, our "unnecessary function definition" anti-pattern should be removed for small functions, but named functions may still be preferred in light of a future modification, in which they may need to be invoked multiple times.

It is worth mentioning that in recent releases, Java has added language-level support for functional programming (e.g., lambda expressions and streams[3]) and other features that promote conciseness (e.g., declaring local variables with `var`). As these features become thoroughly integrated with the language, Java-first programmers would learn more concise coding idioms. When these programmers learn their subsequent languages, including Python, they would be also more likely to write concise idiomatic code. However, the insights presented herein are reflective of what we observed by examining the current programming practices of aspiring Python programmers who had been introduced to programming in Java.

The report findings is a work in progress, and our primary objective is to identify and name a new recurring problem as a call for action for computing educators. The teaching of conciseness should be pursued as a viable pedagogical objective, even for introductory students. Developing a professional programming style should involve striving to write concise idiomatic code. Introductory students need guidance regarding the adherence of their programming assignments to solid

[2]Lambdas were introduced to one of the latest Java releases and may not yet have been in wide use.

[3]https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

software engineering principles, which should include not only coherent design and correctness, but also following a concise and idiomatic coding style. When introducing a new language to students who learned programming in a different language, educators should point out the idiomatic differences between the languages, with a particular emphasis on conciseness. Even though Java has embraced functional programming constructs with their declarative concise nature, much of Java code remains more verbose than Python. So when teaching Java programmers Python, educators may find useful our Java-like verbosity anti-patterns and use them as teaching material.

5 Related Work

A study performed by Chen et al. demonstrates how the first learning programming language can impact the performance and attitudes of students in subsequent programming courses[2]. Michigan State University keeps changing the first programming languages, taught in the university's CS1 courses, in order to enable students to develop up-to-date software development skills, such as web programming[11]. When it comes to differentiating between different semantic representations of the same concept as well as switching from Java to Python and vice versa, students have been observed to find such semantic transfer challenging[13].

When learning a new language, programmers encounter many challenges, such as mixing up the syntax and concepts with their previous programming languages, as reported by Shrestha et al.[10]. Study by Horschig et al. shows that programmers who are proficient in Java or C++ can actually become better Python programmers when it comes to commonly accepted and object-oriented best practices, but not necessarily with respect to Python-specific idioms, such as semicolons and indentation[5].

Programmers who are familiar with Python, but less so with the R language have been found to be able to effectively learn R, when assisted with a guide that explains R programming concepts by means of equivalent Python code[9]. Srinath provides a catalog of distinctive and unique features of Python that may not have equivalents in other languages. [12].

This work builds upon the insights uncovered by these prior studies. Our approach differs in that we focus on formulating anti-patterns, with a particular emphasis on code verbosity caused by prior experiences with Java programming. By formulating our anti-patterns, we aim at providing actionable information that can guide programming tool builders and computing educators, as a way to promote the concise and idiomatic programming style to which Python owes much of its popularity.

6 Conclusions

In this paper, we discussed some code quality issues that arise when Java programmers switch to programming in Python. Because Python is more concise than Java, such programmers, whose programming habits are influenced by their first language, tend to replicate more verbose Java idioms rather than using their native Python counterparts. Specifically, by combing through programming assignments submitted in a class that introduced Java programmers to Python, we identified and described six *Java-like verbosity anti-patterns* that manifested themselves in Python code. We then conducted a pilot study to check whether some of these anti-patterns are also present in open-source Python codebases. Based on our observations, we presented a call for action to computing educators. Computing students should be encouraged to make it a habit of writing concise and idiomatic code in any language.

Acknowledgments

This research was supported in part by NSF through the grants #1744722 and 1712131.

References

[1] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura. 2017. BlockPy: An Open Access Data-Science Environment for Introductory Programmers. *Computer* 50, 5 (2017), 18–26. https://doi.org/10.1109/MC.2017.132

[2] Chen Chen, Paulina Haduong, Karen Brennan, Gerhard Sonnert, and Philip Sadler. 2019. The effects of first programming language on college students' computing attitude and achievement: a comparison of graphical and textual languages. *Computer Science Education* 29, 1 (2019), 23–48. https://doi.org/10.1080/08993408.2018.1547564 arXiv:https://doi.org/10.1080/08993408.2018.1547564

[3] Brian Eastwood. 2020. The 10 Most Popular Programming Languages to Learn in 2020. https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/.

[4] O. Ezenwoye. 2018. What Language? – The Choice of an Introductory Programming Language. In *2018 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE.2018.8658592

[5] Siegfried Horschig, Toni Mattis, and Robert Hirschfeld. 2018. Do Java Programmers Write Better Python? Studying off-Language Code Quality on GitHub. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 127–134. https://doi.org/10.1145/3191697.3214341

[6] Chris F Kemerer. 1995. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering* 1, 1 (1995), 1–22.

[7] S. Nanz and C. A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 778–788. https://doi.org/10.1109/ICSE.2015.90

[8] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information* 11, 4 (2020), 193.

[9] N. Shrestha, T. Barik, and C. Parnin. 2018. It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge. In *2018*

IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). 177–185. https://doi.org/10.1109/VLHCC.2018.8506508

[10] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE*.

[11] Robert M. Siegfried, Diane Liporace, and Katherine G. Herbert-Berger. 2019. What Can the Reid List of First Programming Languages Teach Us About Teaching CS1?. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1256–1257. https://doi.org/10.1145/3287324.3293830

[12] KR Srinath. 2017. Python–The Fastest Growing Programming Language. *International Research Journal of Engineering and Technology (IRJET)* 4, 12 (2017), 354–357.

[13] Ethel Tshukudu and Quintin Cutts. 2020. Understanding Conceptual Transfer for Students Learning New Programming Languages. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 227–237. https://doi.org/10.1145/3372782.3406270