A Study on the Pythonic Functional Constructs' Understandability

Cyrine Zid
Polytechnique Montréal
Montréal, Québec, Canada

Fiorella Zampetti
University of Sannio
Benevento, Italy

Giuliano Antoniol
Polytechnique Montréal
Montréal, Québec, Canada

Massimiliano Di Penta
University of Sannio
Benevento, Italy

ABSTRACT

The use of functional constructs in programming languages such as Python has been advocated to help write more concise source code, improve parallelization, and reduce side effects. Nevertheless, their usage could lead to understandability issues. This paper reports the results of a controlled experiment conducted with 209 developers to assess the understandability of given Pythonic functional constructs—namely lambdas, comprehensions, and map/reduce/filter functions—if compared to their procedural alternatives. To address the study's goal, we asked developers to modify code using functional constructs or not, to compare the understandability of different implementations, and to provide insights about when and where it is preferable to use such functional constructs. Results of the study indicate that code snippets with lambdas are more straightforward to modify than the procedural alternatives. However, this is not the case for comprehension. Regarding the perceived understandability, code snippets relying on procedural implementations are considered more readable than their functional alternatives. Last but not least, while functional constructs may help write compact code, improving maintainability and performance, they are considered hard to debug. Our results can lead to better education in using functional constructs, prioritizing quality assurance activities, and enhancing tool support for developers.

CCS CONCEPTS

• **Software and its engineering → Language features**.

KEYWORDS

Functional Programming, Python, Program Comprehension, Empirical Study

ACM Reference Format:
Cyrine Zid, Fiorella Zampetti, Giuliano Antoniol, and Massimiliano Di Penta. 2024. A Study on the Pythonic Functional Constructs' Understandability. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639211

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3597503.3639211

1 INTRODUCTION

Widely-used programming languages have been conceived to be used by following either the imperative or object-oriented paradigm, *e.g.,* C or Java. Some of these programming languages have also adopted constructs inspired by functional programming. This is the case of Java with the introduction of lambda functions in Java 1.8, Javascript with the introduction of higher-order functions, and, last but not least, Python. Such languages are not intrinsically functional (like Haskell or Scala), but give developers the flexibility to use functional constructs.

Functional constructs have been advocated to help parallelization, avoiding the introduction of side effects in the source code [12, 41]. At the same time, the understandability of such constructs has been questioned and discussed in previous studies and gray literature [13, 23, 26, 39].

Let us focus on Python, which is, to date, Github's second most used language [10]. Previous research has found that changes to some Pythonic functional constructs (lambdas, comprehensions, and map/reduce/filters) induce more fixes than other changes [44].

Despite the discussions in gray literature and the results by Zampetti *et al.* [44], there is a lack of empirical evidence about the extent to which developers encounter more or less trouble when using functional constructs than when using their procedural alternative. To bridge this gap, this paper reports the results of a controlled experiment conducted with 209 paid developers, investigating the extent to which developers encounter difficulties while understanding and modifying code snippets involving functional constructs, as opposed to their procedural alternatives. The study focuses on lambdas, comprehensions, and map/reduce/filter (MRF) functions, as in previous work [44]. While, as pointed out by previous work, there are other Pythonic functional constructs one could consider, we preferred to focus on these because of their wide usage in Python [1, 6], and because of the inherent task complexity limitations when conducting a controlled experiment. The 209 developers have been recruited through the Prolific [15] platform. Specifically, each participant in the experiment had to:

(1) Perform change tasks on functional, as well as procedural code (for which somebody else received the functional alternative);
(2) Provide a perceived, comparative level of understandability over code snippets written in both fashions; and
(3) Provide qualitative insights about the rationale for using (or not) the studied functional constructs in everyday development tasks.

Findings of the study highlight that: (i) when modifying existing code snippets, developers perform better with lambdas than with

their procedural alternatives; instead, comprehensions make change tasks worse than their procedural alternatives; (ii) procedural code is generally perceived as more readable than its functional alternative; and (iii) developers found functional constructs useful to shorten the source code, even if they are challenging to debug.

Results of the study provide empirical evidence on the difficulties experienced by developers when using functional constructs, also depending on their type and complexity. This has implications for developers' education, prioritizing quality assurance on portions of code involving certain constructs, and aiding developers to properly use certain, complex-to-use constructs.

2 THE STUDIED FUNCTIONAL CONSTRUCTS

This section overviews the functional constructs we studied.

```
add = lambda x, y: x + y
print(add(3,5))
```

Listing 1: Lambda expression example.

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
```

Listing 2: List comprehension example.

```
def square(x):
    return x ** 2

def multiply(x, y):
    return x * y

numbers = [1, 2, 3, 4, 5, 6, 7]
squared_numbers = map(square, numbers)
product = reduce(multiply, numbers)
```

Listing 3: Map and Reduce functions examples.

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(is_even, numbers)
```

Listing 4: Filter function example.

Lambda Functions. A lambda function is an anonymous function that can be defined on the fly in the code (as in Listing 1). It can accept any number of arguments, though it is constrained to contain only a single expression. A lambda expression in Python is defined as:

`lambda arguments: expression`.

List Comprehensions. A list comprehension iterates over a list of objects, checks for a specified condition (if any), and computes an expression. This computed value becomes an element in the newly created collection (as shown in Listing 2). A list comprehension can be defined with the following syntax:

`[expression for item in iterable if condition]`

Map/Reduce/Filter Functions. `Map()`, `reduce()` and `filter()`, are three high-order Python functions. They take a function as a parameter and return a function. They have a similar syntax:

`FUN (function, iterable)`

where FUN is either `map`, `filter`, or `reduce`.

`map()` applies a given function to all items in an iterable (*e.g.,* a list) and returns an iterator storing the obtained values (as in Listing 3 where it returns the square of each element).

`reduce()` systematically applies a function (requiring two arguments) to the items within an iterable, progressively producing a singular value. The initial application is carried out on the first two elements of the iterable, followed by applications involving the partial result and the subsequent element. As shown in Listing 3, the `reduce()` function calculates the product of the values contained in the `numbers` iterable.

`filter()` applies a given function to each element of the iterable, producing a Boolean result, keeping only the elements for which the return value is `True`. As an example, in the Listing 4, the filter function retains only the even numbers from the list.

3 STUDY DEFINITION AND PLANNING

The *object* of this study are Pythonic functional constructs, and, specifically, lambdas, comprehensions, and MRF. The *quality focus* refers to their understandability, especially during a change task. The *perspective* is of researchers interested in distilling guidelines for developers, educators, and researchers/tool builders to facilitate the comprehension and maintenance of Python code. The *context* consists of 209 paid developers recruited through the Prolific platform, and code snippets written using functional constructs, as well as their procedural alternatives. The study aims to address the following three research questions:

(1) **RQ$_1$:** *How difficult is it to modify code snippets relying on functional constructs compared to their procedural alternatives?* This research question aims to assess, through change tasks, the understandability of code written using functional constructs versus their procedural alternatives.
(2) **RQ$_2$:** *To what extent do functional constructs influence the perceived source code understandability?* Here we assess the perceived understandability of a code snippet using a functional construct versus its procedural alternative. This is achieved by showing two functionally equivalent code snippets written with and without functional constructs.
(3) **RQ$_3$:** *Why do developers use functional constructs, and in which scenarios?* The last research question qualitatively investigates the reasons why and when developers use functional constructs, but also the perceived disadvantages.

3.1 Context Selection - Study Participants

The study was conducted by leveraging 209 paid developers recruited using the Prolific [15] platform. As highlighted by Tahaei and Vaniea [40], this is the only platform that could involve an adequate number of participants who pass the programming questions. Note that Prolific has been previously used in software engineering research on defect prediction [17], or tool evaluation [32]. Existing studies [7, 33, 35] highlighted the "vague meaning of programming experience". Therefore, we have complemented a set of constraints on the Prolific platform, *e.g.,* in terms of education, skills, and working experience to recruit developers, with further filters, *e.g.,* filtering out all respondents not able to complete any of the six change tasks. Specifically, we have selected the participants on several

requirements: (i) the subject of study was computer science, (ii) they declared to know computer programming, (iii) Python was listed as one of the known programming languages, and (iv) for the education level, we limited our choices to technical/community college, undergraduate (*e.g.,* bachelor), graduate (*e.g.,* master), or doctorate degree.

Given that we estimated a task duration between 40 minutes and 1 hour, and considering our available budget, we offered a payment of 12 UK Pounds-hour (the minimum as per Prolific's policy is 8 UK Pounds-hour).

3.2 Context Selection - Study Objects

Since our experiment shows code snippets to developers and asks them to perform certain tasks, we need to properly select representative ones. To avoid artificial code snippets, we started identifying random instances of the investigated functional constructs from an existing dataset [44]. The selection was made to select, for each type of functional construct, a set of code elements with varying complexity. The complexity is computed as McCabe's cyclomatic complexity [24], by adding one to the number of conditionals, iterations, and generators.

In particular, we picked: 10 lambda usages with a complexity between 1 and 4; 10 comprehension usages, with a complexity between 2 and 6; and 3 map, 3 reduce, and 3 filter usages.

After the selection, we isolated each code snippet from its surrounding context, by creating constant initialization for the used variables. Then, for each code example, one author produced the procedural alternative, that has been double-checked by a different author, both having at least three years of Python programming experience. Specifically, we replaced lambda with an explicit function definition, while comprehensions have been replaced by nested loops and conditionals. For MRF, we replaced their invocation with the explicit implementation of the required operation, *e.g.,* iterating through all the elements of a list and applying to all of them the desired operation. As a last step, we made sure that each code snippet produced an output so that it was possible to objectively assess its outcome, as well as we verified that, given the same input, the output of the functional code snippet was perfectly similar to the one obtained with the "hand-crafted" procedural alternative.

3.3 Dependent and independent variables

The study *dependent variable* is the code understandability, measured as change task correctness for RQ_1, and perceived understandability for RQ_2. The *independent variable* is the use of a functional construct, as opposed to its procedural alternative. While the analysis has been done on multiple constructs (lambdas, comprehensions, and MRF), this is not a variable of the study, as we do not directly compare constructs with each other. In addition, we account for *co-factors* related to constructs' complexity and developers' characteristics. In the following section, we further detail how variables are measured, and what co-factors are.

3.4 Experiment Design and Planning

The experiment consists of a series of questions, asked to each participant through a Web-based questionnaire (a Google form). The questionnaire was organized into 4 sections.

```
lst=[[[i + j + k for k in range(4, 7)] for j in
    range(1, 4)] for i in range(2, 5)]
print(lst)
```

Code snippet with functional constructs: Group 6

Question: Please modify the code so that it will generate a matrix where the elements in the rows are summed if the indexes are all even, otherwise are multiplied.

Figure 1: Example of change task for list comprehensions

```
lst=[]

for i in range(2, 5):
  lt=[]
  for j in range(1, 4):
    l=[]
    for k in range(4, 7):
      l.append(i + j + k )
    lt.append(l)
  lst.append(lt)

print(lst)
```

Code snippet with procedural constructs: Group 5

Question: Please modify the code so that it will generate a matrix where the elements in the rows are summed if the indexes are all even, otherwise are multiplied.

Figure 2: Example of change task for the procedural counterpart of the list comprehension reported in Figure 1

```
i = [1, 2, 3, 4, 5, 6]

def function(i):
    return tuple((-1 if j is None else j for j in i[1:4]))

print(function(i))
```

```
i = [1, 2, 3, 4, 5, 6]
print((lambda i: tuple((-1 if j is None else j for j in i[1:4])))(i))
```

Please select a choice:

1. The first code is definitely easier to understand than the second code
2. The first code is slightly easier to understand than the second code
3. There are no differences in terms of understandability
4. The second code is slightly easier to understand than the first code
5. The second code is definitely easier to understand than the first code

Figure 3: Example of understandability task for lambda and its procedural alternative

In the **first section**, we assess the participants' understanding of code snippets by letting them perform six change tasks. This has been previously done in other studies on program comprehension (*e.g.,* [34, 36, 43]), allowing for an objective evaluation, and better reflecting a scenario in which a developer needs to understand a code snippet during evolution tasks. Specifically, we show a code snippet and ask participants to modify it. Three tasks include code snippets using functional constructs: one with lambda, one with comprehension, and one with map/reduce/filter. The other three tasks (interleaved with the former) feature procedural alternatives of functional code that different participants are changing. Figure 1

Table 1: Assignment of change and comparison tasks across the ten experimental groups. MRF means map/reduce/filter. Note that for each task, there is the identifier of the task, followed (for the change tasks) by a letter indicating whether the snippet used the (F)unctional or (P)rocedural paradigm, and the complexity in parenthesis (for lambda and comprehension only).

Group	Change Tasks						Comparison Tasks		
	Lambda_1	Lambda_2	Comp_1	Comp_2	MRF_1	MRF_2	Lambda	Comp.	MRF
1	lambda-a-F (1)	lambda-g-P (4)	comp-a-F (2)	comp-f-P (3)	filter-a-F	map-c-P	lambda-h (2)	comp-b-(4)	reduce-a
2	lambda-b-P (2)	lambda-g-F (4)	comp-a-P (2)	comp-f-F (3)	filter-a-P	map-c-F	lambda-c (3)	comp-d (5)	reduce-c
3	lambda-a-P (1)	lambda-h-F (2)	comp-b-P (4)	comp-g-F (2)	map-a-F	reduce-b-P	lambda-g (4)	comp-f (3)	filter-a
4	lambda-b-F (2)	lambda-i-P (3)	comp-b-F (4)	comp-g-P (2)	map-a-P	reduce-b-F	lambda-d (2)	comp-a (2)	filter-c
5	lambda-c-F (3)	lambda-h-P (2)	comp-c-F (2)	comp-h-P (4)	reduce-a-P	filter-b-F	lambda-b (2)	comp-j (6)	map-c
6	lambda-c-P (3)	lambda-d-F (2)	comp-c-P (2)	comp-h-F (4)	reduce-a-F	filter-b-P	lambda-i (3)	comp-g (2)	map-a
7	lambda-d-P (2)	lambda-i-F (3)	comp-d-P (5)	comp-i-F (3)	map-b-F	filter-c-P	lambda-a (1)	comp-h (4)	reduce-b
8	lambda-e-F (3)	lambda-j-P (2)	comp-d-F (5)	comp-i-P (3)	map-b-P	filter-c-F	lambda-f (1)	comp-e (3)	reduce-a
9	lambda-e-P (3)	lambda-f-F (1)	comp-e-P (3)	comp-j-F (6)	filter-b-F	reduce-c-P	lambda-e (3)	comp-c (2)	map-b
10	lambda-f-P (1)	lambda-j-F (2)	comp-e-F (3)	comp-j-P (6)	filter-b-P	reduce-c-F	lambda-e (3)	comp-i (3)	map-c

illustrates an example of a change task for list comprehension (assigned to group 6) while its procedural alternative, assigned to a different experimental group (group 5), is depicted in Figure 2.

The **second section** consists of three questions asking the perceived ease of understandability of a functional construct and its (functionally equivalent) procedural alternative. Each participant receives three pairs of code snippets, one related to lambda, one to comprehension, and one to MRF. We ask participants to perform the assessment using a 5-level Likert scale [28], as illustrated in Figure 3. To mitigate possible ordering bias, we change the order in which we show the code snippets: some participants receive the functional before the procedural, and vice versa. Furthermore, to mitigate risks due to "random" answers, we also ask the participants to describe the behavior of the code snippet, so we were sure they understood its meaning.

The **third section** consists of four questions asking the reasons for using functional constructs, as well as the procedural alternatives, *e.g., "If you use lambda functions regularly, can you please explain the reasons why you are using them compared to their procedural alternatives? Please use None as an answer in case you do not use them regularly."*

The **fourth section** (optional), collects demographics. We ask about (i) the highest degree earned, (ii) the current working position, (iii) the years of development experience, (iv) and, the frequency of usage of each of the considered functional constructs by providing as options: Never; Less than 25% of the development tasks; Between 25% and 75% of the development tasks; and More than 75% of the development tasks. The declared usage frequency will be used as one of the criteria to discard responses.

Table 1 summarizes the assignment of change (RQ_1) and comparison (RQ_2) tasks to the study participants. For the change tasks, we indicate the type of construct being studied, whether we show the (F)unctional construct or its (P)rocedural alternative, and, for lambda and comprehension, the complexity of the construct within the code snippet (in parenthesis). As the table shows, we have 10 experimental groups, designed with the following constraints:

- For lambda and comprehension, participants in the same group receive code snippets with constructs of different complexity. As MRF mainly apply a function (which from the client side may even be a black box) to each data structure entry, complexity was not considered a relevant factor, so we have not considered their complexity in both the experimental design and the analysis methodology.
- The comparison tasks are designed by showing the functional before the procedural or vice versa to different experimental groups.
- For MRF, the experimental groups are designed so that each participant receives a map example, a reduce example, and a filter example, *i.e.,* one for the first change task, one for the second change task, and one for the comparison.

3.5 Experiment Execution

Before conducting the experiment, its design was submitted to a University Ethical Board for their approval, as the study involved humans. After receiving the approval, we proceeded with the operation, administering the tasks to the study participants. To avoid having the same person in different experimental groups to earn money (this was explicitly disallowed by the protocol we published), we created 10 different Web forms, and with them, 10 sequential "Studies" on Prolific. After each participant completed the questionnaire, one author (i) checked whether the participant had not already participated in a different previous experimental group, and (ii) performed a sanity check to guarantee questions were answered. If such checks passed, the author approved the payment, otherwise, the participant was discarded. Where necessary (*i.e.,* having fewer participants), we extended the invitation so that we ended up with at least 20 answers for each experimental group. Out of 248 returned questionnaires, we discarded 39, of which 18 did not pass the sanity check, 14 have never used any of the studied functional constructs, and 7 did not provide any correct answer for all six change tasks.

3.6 Results Collection and Processing

After the collection of the results from the 10 experimental groups, we analyzed the answers. First, we excluded participants who declared having never used any of the considered functional constructs. However, we kept those that only used some constructs, *e.g.,* lambda and comprehension, but they are only considered for the questions related to that construct.

For the **change tasks**, each code snippet produced by the participants was scrutinized to check whether the change was implemented and the output was not faked (*e.g.,* the code snippet just produced some code generating the required output). After that, the code snippets were executed through a test script, comparing the scripts' output with the expected one. The outcome of each change task was considered *correct* if the test passed, *wrong* in case of run-time errors or failing test. Using the results of this analysis, we also excluded (from all the analyses of our study) 7 participants who performed all tasks wrongly.

For the **comparison tasks**, there are two aspects to account for, namely the description of the functionality implemented by the showed code snippet, and the perceived understandability. While the latter was trivial to collect and analyze, for the former, two authors cooperatively scrutinized the responses describing the code snippet's behavior. Each response was classified into four categories: (i) correct, (ii) somewhat correct (*e.g.,* some details about the behavior were missing, but, overall, the respondent understood the general behavior), (iii) wrong, and (iv) automatically generated. For the latter, the authors used GPTZero [27] to check whether the produced text was likely to be AI-generated. We have considered this to be the case if the outcome of GPTZero reads: *"Your text is likely to be written entirely by AI"*.

For the **reasons for using functional constructs, as well as their procedural alternatives**, we applied open card-sorting [38] on the provided answers. Also in this case, we excluded all answers likely to be AI-generated based on the outcome of GPTZero [27]. On the remaining answers, two authors (annotators in the following) performed a joint categorization of 20 responses to agree on the labeling criteria. After that, they used an online spreadsheet to independently categorize the remaining responses. Specifically, the annotators could select a category from a list of possible ones. When a new category was required, the annotator could add it to the list, so it became available to both of them. Note the sheet was designed so that each annotator could assign multiple categories to the same answer when a respondent provided multiple reasons for using a construct. To assess the quality of the manual labeling, we used Krippendorff's α [19] reliability coefficient. This procedure can handle missing codes, as well as the presence of multiple categories assigned to the same answer. The two annotators achieved an $\alpha = 0.87$ (very strong) concerning the reasons for using functional constructs, and $\alpha = 0.70$ (strong) for reasons behind preferring the procedural alternatives.

3.7 Analysis Methodology

In the following, we describe the methodology used to address the three research questions.

To address $\mathbf{RQ}_1$, for each functional construct, we exclude data points for respondents who never used the construct. This left 160 valid responses for lambdas, 192 for comprehensions, and 159 for MRF. After that, we statistically analyze the remaining data points.

Specifically, to perform the statistical analysis of the results, we need to use a procedure that tests the relationship between the correctness of the change task and various factors besides the main experimental factor. Also, the procedure needs to check the longitudinal, within-subjects effect. To this aim, we use a mixed-effect logistic model, leveraging the *glmer* function of the *R lme4* package [2]. We use such a model because it is suitable for an experiment with dependent samples. More in detail, the participants (through their identifiers) are the model's random effect, the dependent variable is the change task correctness, and the independent variables are:

- **Main Factor:** Whether the code snippet was shown with the *functional* construct or with its *procedural* alternative;
- **Complexity:** The effect of the constructs' complexity (for lambdas and comprehensions only) and its interaction with the main factor;
- **Approvals:** The participants' number of previous task approvals on Prolific;
- **Usage Frequency:** The participant self-declared frequency usage for that type of construct;
- **Student:** Whether the participant is currently a student, as declared on Prolific.

Since the *glmer* procedure involves multiple factors for which p-values are computed, we adjust them using the Benjamini-Hochberg procedure [4].

As the experiment design foresees, for MRF, the longitudinal analysis with the *glmer* model will only be possible by ignoring the construct type. The latter is mainly due to having only two change tasks, one with one of the three functions and a different one with a different function. For this reason, for MRF, we perform an additional (unpaired, between subjects) analysis for each specific function through a logistic generalized linear model (*glm*). We complement the test with the Odds Ratio (OR) effect size measure, where the OR reads as the increasing odds to flip the dependent variable to *true* (*i.e.,* correct answer) for a unitary increase of the independent variable, or, for categorical dependent variables, the increasing odds when the variable assumes that value.

To address $\mathbf{RQ}_2$, for each functional construct, we report the comparative, perceived level of understandability using diverging stacked bar charts. Furthermore, we show the relationship between the results and (i) the declared frequency of usage of the different functional constructs, as well as (ii) the construct complexity (for lambda and comprehension only). This is done with an ordinal logistic regression model, implemented through the *polr* function of the *R MASS* package [42]. In this case, the OR indicates how the unitary increment of an independent variable increases the odds of a unitary increase of the dependent variable. Also in this case, p-values are adjusted with the Benjamini-Hochberg procedure [4].

Finally, to address $\mathbf{RQ}_3$, we report and discuss the elicited reasons for using functional constructs, as well as their procedural alternatives.

4 STUDY RESULTS

In the following, after reporting the demographics of the study participants, we report and discuss the study results.

4.1 Participants' Demographics

Although demographic questions were optional, all participants answered them. In terms of highest education qualification, 122 out of 209 participants hold a bachelor's degree, 41 have a master's degree, 4 have received a Ph.D., and the remaining 42 have a high

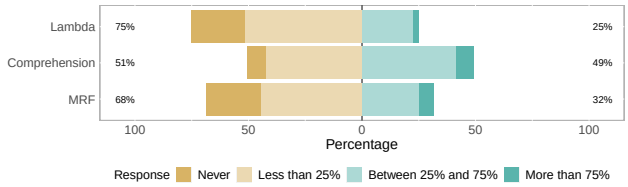

Figure 4: Declared construct usage.

Table 2: Number and percentage of correct/wrong tasks for functional constructs and their procedural alternatives

Construct	Functional		% Corr.	Procedural		% Corr
	Corr.	Wrong		Corr.	Wrong	
Lambda	112	98	53.33	115	95	54.76
Compr.	99	111	47.14	114	96	54.29
MRF	80	130	38.10	85	125	40.48

Table 3: RQ$_1$: Mixed-effect logistic regression relating the use of lambdas with the correctness of the change task

AIC	BIC	logLik	deviance	df.residuals	
438.3	468.5	-211.2	422.3	314	
SCALED RESIDUALS:					
Min	**1Q**	**Median**	**3Q**	**Max**	
-1.98	-0.95	0.49	0.85	1.44	
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
User (Intercept)				0.2299	0.4795
Number of obs: 322, groups: User, 160					
FIXED EFFECTS:					
	OR	**Estimate**	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	3.73	1.32	1.12	1.17	0.24
MainFactorProc	0.11	-2.19	0.67	-3.29	**<0.01**
Compl.	0.75	-0.29	0.18	-1.55	0.24
Usage Freq.	1.32	0.27	0.23	1.20	0.24
Approvals	1.00	-0.00	0.00	-1.41	0.14
StudentTrue	0.35	-1.04	0.88	-1.18	0.24
MainFactorProc:Compl.	2.67	0.98	0.28	3.56	**<0.01**

school diploma while being undergraduates. For what concerns development experience, 26 out of 209 participants declare to have between 5 and 10 years of development experience, while 183 have less than 5 years of experience. Hence, our sample is, in its majority, representative of junior/average-seniority developers.

Figure 4 summarizes the declared usage of the functional constructs object of the study. As the figure shows, 49% of the participants use comprehensions for over 25% of their tasks, while for MRF this happens only for 32% of the participants, and for lambda only for 25% of them.

4.2 RQ$_1$: How difficult is it to modify code snippets relying on functional constructs compared to their procedural alternatives?

Table 2 reports the number of correct and wrong tasks for functional constructs and their procedural alternatives. Looking at it, we can only state that for comprehension, the percentage of correct answers is lower (47.14%) for the functional than for the procedural

Table 4: RQ$_1$: Mixed-effect logistic regression relating the use of comprehensions with the correctness of the change task

AIC	BIC	logLik	deviance	df.residuals	
534.7	566.3	-259.3	518.7	378	
SCALED RESIDUALS:					
Min	**1Q**	**Median**	**3Q**	**Max**	
-1.64	-0.96	0.62	0.99	1.59	
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
User (Intercept)				0	0
Number of obs: 386, groups: User, 192					
FIXED EFFECTS:					
	OR	**Estimate**	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	1.15	0.14	0.89	0.16	0.88
MainFactorProc	6.11	1.81	0.60	3.00	**0.02**
Compl.	1.02	0.02	0.11	0.15	0.88
Usage Freq.	0.86	-0.15	0.16	-0.94	0.61
Approvals	1.00	-0.00	0.00	-1.12	0.61
StudentTrue	1.19	0.18	0.66	0.27	0.88
MainFactorProc:Compl.	0.65	-0.43	0.17	-2.58	**0.03**

Table 5: RQ$_1$: Mixed-effect logistic regression relating the use of MRF with the correctness of the change task

AIC	BIC	logLik	deviance	df.residuals	
443.0	465.6	-215.5	431.0	314	
SCALED RESIDUALS:					
Min	**1Q**	**Median**	**3Q**	**Max**	
-0.92	-0.89	-0.76	1.12	1.52	
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
User (Intercept)				0	0
Number of obs: 320, groups: User, 159					
FIXED EFFECTS:					
	OR	**Estimate**	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	1.64	0.50	0.96	0.52	0.93
MainFactorProc	0.95	-0.05	0.23	-0.23	0.93
Usage Freq.	0.74	-0.30	0.18	-1.60	0.55
Approvals	1.00	-0.00	0.00	-0.29	0.93
StudentTrue	0.94	-0.07	0.76	-0.09	0.93

alternative (54.29%). For lambda and MRF, the difference is again in favor of the procedural, yet the difference is 1-2% only.

Table 3 shows the results of the mixed-effect model for the lambda construct. As the table shows:

(1) There is a statistically significant effect of the main factor. Specifically, giving procedural code, in general, reduces the odds of $OR = 0.11$ to implement a correct change;
(2) The complexity alone does not play a statistically significant effect;
(3) There is a positive statistically significant interaction between the main factor and the complexity of the construct. Specifically, if the participant receives procedural code, increasing its complexity increases 2.67 times the odds of correctly implementing the change, compared to those who receive its functional alternative;
(4) Participant-related metrics (Usage Frequency, Approvals, and Student status) do not play a statistically significant role.

Overall, we can conclude that when the complexity of the lambda increases, it is easier to correctly modify the code snippet by operating on its procedural alternative, *i.e.,* modify a separate function.

Table 6: RQ_1: Logistic regression relating the use of map with the correctness of the change task (AIC=121)

	Estimate	Std.Error	z value	Pr(>|z|)
(Intercept)	16.48	1370.51	0.01	0.99
MainFactorProc	0.67	0.48	1.39	0.82
Usage Freq.	-0.03	0.38	-0.09	0.99
Approvals	0.00	0.00	-0.07	0.99
StudentTrue	-15.75	1370.51	-0.01	0.99

Table 7: RQ_1: Logistic regression relating the use of reduce with the correctness of the change task (AIC=118)

	Estimate	Std.Error	z value	Pr(>|z|)
(Intercept)	-13.20	1696.36	-0.01	0.99
MainFactorProc	-0.26	0.47	-0.56	0.99
Usage Freq.	-0.83	0.44	-1.87	0.30
Approvals	0.00	0.00	-0.36	0.99
StudentTrue	14.38	1696.36	0.01	0.99

Table 8: RQ_1: Logistic regression relating the use of filter with the correctness of the change task (AIC=157)

	Estimate	Std.Error	z value	Pr(>|z|)
(Intercept)	-15.80	1383.44	0.01	0.99
MainFactorProc	-0.06	0.40	-0.16	0.99
Usage Freq.	-0.18	0.33	-0.54	0.99
Approvals	0.00	0.00	-0.76	0.99
StudentTrue	15.56	1383.44	0.01	0.99

Table 4 shows the results of the mixed-effect model for the comprehension construct. In this case:

(1) The main factor has a statistically significant effect. However, in this case, giving procedural code increases the odds of correctly implementing the change ($OR = 6.11$);
(2) The complexity alone does not have a statistically significant effect;
(3) The main factor has a statistically significant interaction with the complexity, yet the effect is the opposite of what was reported for lambdas. When procedural code is used, an increase of complexity reduces the odds of $OR = 0.65$ for correctly implementing the change. Possibly, developers better inspect loops and conditionals in a comprehension expression than in a (longer) procedural code.
(4) Participant-related metrics do not play a statistically significant role.

For MRF (see Table 5) we were unable to observe any statistically significant difference. However, as mentioned before, the paired analysis is not sufficient for MRF, because participants received, over the two change tasks, code snippets dealing with different functions.

Therefore, we performed an unpaired analysis for each function whose results (see Tables 6, 7, 8) did not report any statistically significant difference. A possible reason could be the relative simplicity of the functions called by the MRF (needed by the controlled experiment setting), and, consequently, of their procedural alternative.

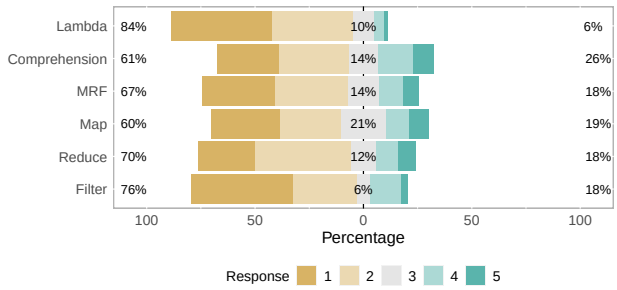

Figure 5: Perceived understandability of functional constructs, compared to their procedural alternative (low values indicate a preference towards the procedural)

Table 9: RQ_2: Ordinal logistic regression results of the relationship between perceived understandability of the functional constructs and (i) participants' usage frequency, and (ii) constructs' complexity (except for MRF)

Lambda (90 data points)					
	OR	**Estimate**	**StdError**	**t-value**	**p-value**
Usage Freq.	2.29	0.83	0.31	2.65	**0.02**
Compl.	0.88	-0.13	0.18	-0.73	0.47
Comprehension (120 data points)					
	OR	**Estimate**	**StdError**	**t-value**	**p-value**
Usage Freq.	1.49	0.40	0.22	1.84	0.07
Compl.	0.77	-0.26	0.11	-2.27	**0.04**
MRF (103 data points)					
	OR	**Estimate**	**StdError**	**t-value**	**p-value**
Usage Freq.	1.79	0.58	0.25	2.31	**0.02**

RQ_1 Summary: Procedural code is easier to modify than complex lambdas. In contrast, complex comprehensions are easier to modify than procedural code for list construction. For MRF, no statistically significant differences were found.

4.3 RQ_2: To what extent do functional constructs influence the perceived source code understandability?

Figure 5 shows the perceived understandability of different functional constructs compared to their procedural alternatives. Values on the 1-2 side indicate a strong preference towards the procedural alternative, while 4-5 indicate a better-perceived level of understanding for the functional constructs. The study participants almost always prefer the procedural code, and this is particularly true for lambdas, where only 6% of the participants have a better-perceived level of understanding for the functional alternative. As shown in the RQ_1 results, this is the construct for which an increasing level of complexity makes the changes more error-prone. Comprehension is, instead, the construct for which there is a relatively larger percentage of positive answers (26%). Indeed, as found

Table 10: Reasons for using functional and procedural code

Reason	Lambdas	Comp.	MRF	Proc.
Coding time	11	12	10	2
Ease of use	4	10	8	7
Maintainability	14	9	14	6
Performance	7	28	23	6
Readability/Understandability	19	89	33	27
Size	41	76	39	–
Lack of knowledge	–	–	–	16
Project constraints	–	–	–	5
Simplify debugging	–	–	–	9

in the RQ_1 results, when the complexity increases, the study participants made fewer errors when modifying comprehensions than when modifying their procedural alternatives. The percentage of positives for MRF (18%) is in between lambda and comprehension, with small differences between the three functions. In this case, while the perceived level of understanding appears to be in favor of the procedural alternative, this was not reflected by better performance in the change tasks.

Table 9 reports the results of the ordinal logistic regression investigating the relationship between the perceived ease of understanding of the constructs and the construct usage frequency, as well as the constructs' complexity (except for MRF). In all cases, the ease of understanding positively correlates with usage frequency, with statistically significant *p*-values for lambda and MRF, and marginally significant *p*-value for comprehension. The complexity is statistically significant, with a negative effect ($OR = 0.77$) for comprehension, while it is not statistically significant for lambda.

RQ_2 Summary: Functional constructs are, in general, perceived as more challenging to understand than their procedural alternatives. Such developers' perception depends on the constructs' usage frequency, as well as the construct's complexity.

4.4 RQ_3: Why do developers use functional constructs, and in which scenarios?

Table 10 reports the results of the open coding for what concerns the reasons for using lambdas, comprehensions, and MRF, as well as their procedural alternatives. As it can be seen, the first five reasons are shared across both functional constructs and their procedural alternatives. *Size* is specific for using functional constructs, while *Lack of knowledge*, *Project constraints*, and *Simplify debugging* are specific for preferring their procedural alternatives.

In the following, we provide a discussion of the reasons, together with examples of what is being highlighted by our participants.

Coding time. When implementing or enhancing features, developers tend to opt for the paradigm that ensures better productivity. Specifically, for lambdas, respondents mention the saved time instead of defining a function. Of course, the latter is true in all cases where the function does not require to be used more than once in the code, *i.e.,* P_{65} stated *"it's a waste of time to define a function if all I'm ever going to do with it is pass it to another function once"*. As regards the usage of comprehensions, the save in coding time seems, as reported by P_{184}, directly related to the total number of lines to be written *"to gather together a sub-list"*. For MRF, developers motivate their use because as stated by P_{161}, using MRF, it is possible to *"effortlessly reuse a function throughout different sections of your code"*. However, deciding whether to use the functional/procedural alternative is a trade-off between saved coding time and code readability, *i.e., "They save time and don't compromise the readability as much as lambda"* (P_{59}).

As the last column of Table 10 shows, coding time may also be a reason for opting towards the procedural alternative. However, by preferring the procedural alternative, reduced coding time may lead to less readable code, hence they may be preferred when nobody else has to look at the same piece of code, *i.e.,* P_1 stated that *" if I am going to be the only looking at my code I care less about how clean it looks. It takes less time to just manipulate variables directly rather than sending them to a function and getting the output."*

Ease of use. Coding time is not the only way to measure developers' productivity. A different proxy can be related to how simple it would be to deal with the construct, taking into account the task at hand. Specifically, for lambdas, P_{194} stated that *"they are much easier to write and can usually fit on one line"*. About comprehensions, participants highlight the simplicity in translating the logic of the feature to be implemented, *i.e.,* P_{133} stated *"the way I think can be easily translated into the code"*. For MRF, developers recognize the simplicity in properly applying the same operation *"to all members of a list.'* (P_{120}). Ease of use, is also a reason for opting towards the procedural alternative, *e.g.,* P_{44} reported: *"it's sometimes easier to write code idea straightforward than making it short and clean"*.

Maintainability. For what concerns lambdas, developers mention the simplicity in changing them when needed, as well as the reduction of redundant code due to creating functions, *e.g., "they reduce boilerplate"* (P_{31}). The latter also applies to comprehensions, for which P_{195} highlights that *"they can help to reduce code duplication and make the code more maintainable"*. Furthermore, P_{180} mentions the reduced defect-proneness *"mak[e] the code more predictable, easier to test, and less prone to bugs."* As for MRF, maintenance becomes easier as one avoids writing the same logic over and over again, and, more importantly, *"already implemented constructs behave better than ad-hoc written constructs"* (P_{75}). However, seven respondents point out maintainability reasons in favor of the procedural paradigm. This is the case in scenarios where it is important to have better control of the code under development, as well as flexibility or customization options, *e.g.,* P_{114} highlighted: *"such as when the operation to be performed on each element of a list depends on complex conditions or involves side effects"*.

Performance. Functional programming is mainly intended to avoid side effects, as well as to improve performance [12, 41]. This has been confirmed by our respondents. Specifically, for lambdas, P_{164} mentions that *"They are more efficient, as they can take advantage of the computational resources available on the server and can be executed in parallel"*. For comprehensions, P_{108} reports that they *"can be faster than for loops because they are optimized for this use case in the Python interpreter"*. However, P_{55} highlights that speed matters only when processing large datasets. This is also true for MRF, for which P_{93} states that *"they're useful when working with large iterable since they perform lazy evaluation, which prevents the program from using more memory than needed."* Despite

such developers' perceptions, recent research showed contradictory results in terms of the performance benefits introduced by Pythonic idioms [46]. Finally, five respondents highlight scenarios where procedural may be better for performance. As an example, *"procedural alternatives may be more performant especially for certain types of operations or data structures. This can be important in high-performance or large-scale applications"* (P_{186}).

Readability/Understandability. Readability/understandability may be better for functional constructs, or for their procedural alternatives, based on the task at hand. As an example, for comprehensions, P_{96} states that *"they're more elegant and aren't any less readable in simple cases, for instance when the condition is simple"*. For MRF, there are contradictory opinions. On the one hand, they *"express intent better"* (P_{16}). On the other hand, *"they can quickly do the job they are meant to do, however, they are also possibly confusing"* (P_{122}). Furthermore, developers who prefer procedural code mention that functional constructs *"can be more difficult to read and understand than traditional for loops, especially for developers who are less familiar with functional programming concepts"* (P_{55}).

Size. This category relates to choice due to writing shorter code, which could impact productivity, understandability, and maintainability. As expected, this reason arises only when looking at why developers prefer functional constructs. 71 respondents state this for comprehensions, *e.g.,* P_{126} reports: *"List comprehensions are typically a single line of code which can make them easier to read and understand compared to longer multi-line for loops or map/filter functions."*, yet, also, in this case, there may be trade-offs, *e.g., "it's important for me to use them appropriately and not sacrifice readability for conciseness"* (P_{139}). 35 and 33 respondents mention size as a reason for using lambda, *e.g., "they allow for concise function definitions in a single line of code"* (P_{71}), and MRF, *e.g., "to write more compact code as long as readability doesn't suffer"* (P_{40}).

Lack of knowledge. Sometimes, developers make their decisions based on their level of experience with programming language features. Indeed, 16 respondents admit relying on the procedural paradigm due to *"a lack of familiarity or preference for imperative programming styles"* (P_{143}).

Project constraints. Four respondents state that the decision behind adopting the procedural paradigm is simply dictated by the requirements or constraints of the projects they are contributing. For instance, P_{202} reports that *"non-functional code may be required or preferred in certain environments or domains that do not support functional programming"*, while P_{34} states that *"some projects may require the use of non-functional programming constructs due to compatibility requirements."* One aspect being highlighted by P_{139} deals with the presence of legacy code, where *"If a project has a lot of legacy code that uses non-functional constructs[,] it may be more practical to continue using those constructs, rather than rewriting the code using functional constructs."*

Simplify debugging. Seven respondents highlight preferring the procedural paradigm due to the challenges encountered when debugging "functional" code, also because of the lack of suitable tool support. Indeed, by adopting functional constructs, it is hard to trace the flow of data through multiple operations. For instance, P_{135} reports that *"traditional loops or conditional statements may be easier to debug than functional constructs, as they provide more visibility into the program flow and state."* Furthermore, the difficulty in debugging may become more problematic when sharing code, *e.g., "comprehensions are difficult to debug, so I tend to avoid them when I work in groups"* (P_{172}).

RQ_3 Summary: The main reasons for using functional constructs are writing less code, (perceived) improved performance, and maintainability. However, debugging functional constructs is seen as challenging.

5 IMPLICATIONS

This section discusses the study's implications for researchers/tool builders, practitioners, and educators.

Researchers and Tool builders should improve the support systems for writing, testing, and debugging functional constructs. By enhancing the tools available, developers can be empowered to create more robust and efficient code. To this extent, AI-empowered techniques may support the automated completion, error detection, and even repair of such constructs. For what concerns testing, a big gap is represented by the lack of coverage analysis and debugging tools specific for functional constructs, *e.g.,* loops and conditionals in comprehensions, which makes the quality assurance activity more challenging. Another useful recommendation would be to spot possible performance anti-patterns or, in general, bad smells in functional constructs, hence suggesting appropriate refactoring operations [45]. Furthermore, researchers could delve deeper into the realm of study, employing sophisticated techniques such as eye-tracking technology and other monitoring tools. By conducting in-depth analyses, they can gain profound insights into how developers comprehend and use these constructs. This research might not only enrich our understanding of the cognitive processes involved but also pave the way for the design of even more intuitive and user-friendly programming tools.

Practitioners should ponder the adoption of functional constructs, factoring in the unique project requirements, coding styles employed, and the complexity of the task to implement. Developers have a trade-off: on the one hand, as we found in RQ_3, developers may choose the constructs that better reflect their "thinking" about the piece of functionality to be implemented. On the other hand, if a project follows a coding style with fairly limited use of functional constructs, introducing them through refactoring or when adding new code may not necessarily be beneficial, but it may rather harm software understandability. From a different perspective, practitioners should be mindful of the impact on future maintenance activities, especially for newcomers joining the project. Last, but not least, practitioners should perform specific testing of functional constructs, *e.g.,* by achieving constructs' (branch) coverage. Also, whenever available, developers should use static analysis tools capable of suggesting potential bugs or refactoring related to such constructs. For example, *Pylint* [21] features some checks specific to lambdas and comprehension (*e.g., unnecessary lambda*, *unnecessary comprehension*, or *consider using comprehension*).

Finally, *Educators* should not only teach the syntax of functional constructs, but also train students about their quality assurance, *e.g.,* through dedicated code inspection and test cases, and understanding. Moreover, it may be desirable to outline and discuss different development scenarios in which such constructs could be useful or,

instead, counterproductive, negatively affecting program comprehension.

6 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. The main threat is related to how we measure the understandability of the functional constructs against their procedural alternatives. We have chosen to assess the understandability during a change task (RQ_1), as was done in previous research [34, 36, 43], as well as to ask about the *perceived understandability* in a comparative assessment (RQ_2). It is possible that what we measure might not reflect the achieved level of the construct's understanding. To mitigate this threat, we correlated, for each construct, the *perceived* understandability in terms of ability to perform a change task (RQ_1) with the perceived understandability as declared for the comparison task (RQ_2) using the Kendall's Tau correlation [22]. For all functional constructs, results are never statistically significant (p-value=0.4 or greater), with a negligible correlation coefficient (< 0.05). Based on this analysis, we can state that the perceived understandability does not reflect the correctness of the change task.

Threats to *internal validity* concern factors that could have influenced our results. We controlled for factors such as constructs' complexity, as well as participants' demographics, *i.e.,* the declared level of usage of the construct, the number of approvals on Prolific, and whether the participant is a student. We are aware that there could be other factors, beyond our control, that could influence our results. However, we did our best in keeping the functional code as close as possible to the original code, and translating each code snippet into the equivalent procedural alternative with the same length, complexity, and same identifiers (but loop variables). Finally, for RQ_3, we used GPTZero [27] to check for AI-generated answers, yet, given the study settings, we could not exclude that respondents leveraged the Web to provide us insights.

Threats to *conclusion validity* concern the relationship between the experimentation and outcome. To answer our research questions, wherever appropriate, we used tests suitable to the nature of the observed data (*i.e.,* logistic models, as well as ordinal logistic models). Since multivariate models involve multiple factors, p-values are adjusted with the Benjamini-Hochberg correction [4]. Finally, we checked the reliability of our qualitative analysis by employing a suitable inter-rater agreement measure, *i.e.,* Krippendorff's α [19].

Threats to *external validity* concern the generalizability of our findings. The main threat is due to the representativeness of Prolific workers as real software developers. We have tried to mitigate this threat as much as possible with (i) participants' pre-screening, and (ii) several filters on the collected results. Nevertheless, replications in other contexts, *e.g.,* classrooms or industrial settings, are desirable. In particular, experienced Python developers, well used to such constructs, may exhibit different performance. Truly, it might be difficult to recruit a large number of such participants, and this was our primary rationale for choosing Prolific [15].

7 RELATED WORK

In this section, we discuss related work about (i) functional programming and "Pythonic" constructs, and (ii) program comprehension studies on source code involving human participants.

7.1 Functional programming and "Pythonic" constructs

As stated in the introduction, Python can be used with multiple paradigms. Dyer *et al.* [6] found that existing open-source code is largely object-oriented, with few parts that are mainly functional. After the introduction of lambdas in Java version 8, researchers started investigating how developers rely on lambdas expressions and functional operations, given their ability to enable parallelism and make the code more succinct and readable. Tanaka *et al.* [41] showed that lambda is the most accepted function idiom in Java (16%), while Rao and Chimalakonda [31] found that 78.57% of open-source Python projects have at least one lambda expression in their code.

Alexandru *et al.* [1], conducted an interview-based study on the usage of "Pythonic" idioms, showing that developers with different levels of experience have different perceptions of "Pythonic" idioms in terms of improving source code understandability and performance. Concerning the constructs we study, their work points out the improved understandability and performance for comprehensions, and the improved performance for lambdas.

Lucas *et al.* [23] conducted a study by examining 66 pairs of real code snippets before and after the introduction of lambdas and measuring code readability. Their results did not indicate improved readability due to lambdas. However, when surveying developers, they perceived that lambda expressions tend to enhance program comprehension. Taking a different approach, Hanenberg and Mehlhorn [13] evaluated the readability of lambdas compared to anonymous inner classes (AICs) in Java, showing that lambdas without type annotations are more readable than AICs. In our study, we found that lambdas may generally improve understandability, unless they become too complex, although developers have a better understandability perception towards procedural code.

Zheng *et al.* [47] conducted a mining study to examine the potential impact of collateral side effects on the use of lambdas in Java programs. The study showed that, sometimes, lambdas are factored out to improve code extensibility. Gyori *et al.* [12] proposed `LambdaFicator` [9], an approach aimed to facilitate the conversion of imperative code to functional code using lambdas. On the same line, Zhang *et al.* [45] proposed a more general tool able to convert non-idiomatic code toward 9 types of idioms, including various types of comprehensions. In a follow-up study, Zhang *et al.* [46] studied the performance of such idioms, showing that Python idioms, if used in a realistic development scenario, do not necessarily introduce benefits in terms of performance.

Finally, Zampetti *et al.* [44] conducted a mining study to investigate the extent to which the addition/change of functional constructs has higher odds of inducing fixes than other changes. Their results highlight that lambdas and comprehensions have higher odds of inducing fixes than MRF. On the one hand, our results confirm, in general, the perceived higher level of understandability for procedural alternatives than for functional constructs. On the

other hand, we found that the outcomes of the change tasks vary depending on the construct and its complexity.

7.2 Code comprehension studies with humans

Developers spend a substantial portion of their time comprehending source code while conducting both development and maintenance tasks [37]. This has led several researchers to further investigate developers' behavior during program comprehension tasks. For instance, Hofmeister *et al.* [14] studied the effect of different identifier naming styles on program comprehension during bug localization. Their results show that source code is more complex to comprehend when using only letters and abbreviations as identifier names. Johnson *et al.* [18] conducted a controlled experiment involving 275 participants aimed at assessing the readability of "nesting" and "looping", highlighting that decreasing the level of nesting will decrease the developer's time in properly understanding a code snippet. On the same line, McCauley *et al.* [25] conducted a study involving students to compare their ability to properly comprehend recursive and iterative programs.

Yu *et al.* [43] looked at how developers comprehend test code. By experimenting with 44 developers, they found that the level of experience with automated tests is an influential factor in properly understanding and extending an existing test suite.

A different research thread has looked at "atoms of confusion" in source code, *i.e.,* code elements that tend to negatively impact program comprehension. Gopstein *et al.* [11] conducted a study involving humans to identify 15 atoms of confusion in source code. On the same line, Langhout and Aniche [20], conducted a controlled experiment to identify the most prevalent atoms of confusion in Java source code. de Oliveira *et al.* [5] conducted an eye-tracking study to examine how the presence of atoms of confusion impacts the time developers spend fixating on source code.

We share with all the aforementioned work the goal to study the impact of certain code constructs on program comprehension. To the best of our knowledge, ours is the first study with human participants concerning Pythonic functional constructs.

Some studies leveraged eye tracking to better understand developers' behavior during program comprehension tasks. For instance, Jbara and Feitelson [16] conducted an experiment to measure the time and effort spent by developers reading and understanding regular code, *i.e.,* repetitions of the same basic pattern. Peitek *et al.* [30] used eye-tracking to record the eye movements of 31 developers with different levels of expertise (novices and intermediate developers), to confirm or confute existing knowledge about the effects of linearity of source code on reading order. Bauer *et al.* [3] used an eye-tracker device to shed light on the impact of indentation styles on program comprehension. Specifically, they asked 22 participants to provide the output of code snippets with different indentation styles written in Java. Their results show that indentation does not significantly influence code comprehension.

Other studies leverage an even more complex setting using medical devices. For instance, Peitek *et al.* [29] conducted a study with functional magnetic resonance imaging (fMRI), involving 19 participants to observe the comprehension level of code snippets with different complexity levels. Fakhoury *et al.* [8] used a new methodology consisting of functional Near Infrared Spectroscopy (fNIRS) and eye tracking devices to properly measure program comprehension, *i.e.,* the impact of lexical, structural, and readability issues on developers' cognitive load during bug localization tasks.

8 CONCLUSION AND FUTURE WORK

This paper reported the results of a controlled experiment in which we studied the effect of some functional Pythonic constructs—and specifically lambdas, comprehensions, and map/reduce/filter (MRF)—on program comprehension. The experiment has been conducted with 209 paid developers recruited from *Prolific* and consisted of change tasks, comparative assessment of the perceived understandability, and questions about the developers' reasons for (not) using such constructs. Results of the study show that:

(1) Developers using lambda better performed their change tasks than when using the procedural alternatives, although this is no longer true when complexity increases. The opposite happens for comprehensions, which are generally more error-prone than their procedural alternatives. However, when complexity increases, comprehensions become advantageous, likely because they do not scatter complexity. No statistically significant differences were found for MRF.
(2) Developers generally perceive functional constructs as more challenging to understand than their procedural alternatives. However, this naturally depends on the usage frequency of such constructs and, for comprehensions, on the constructs' complexity.
(3) While developers acknowledge code size reduction as a pros for using functional constructs (without affecting maintainability), they still face challenges when debugging code following the functional paradigm.

In summary, given our participants (with mainly junior or medium-level seniority), results tell that the difficulty of understanding and changing the studied constructs may vary from one construct to the other, and also depend on the constructs' complexity.

Future work aims at replicating the study in other contexts, especially field studies with professionals having extended expertise with Python, as well as experiments in classroom settings, and at assessing the advantages/disadvantages of other types of constructs.

Furthermore, given the qualitative feedback, and the implications stated above, it is desirable to develop suitable tools to better support developers in writing, testing, and debugging such constructs.

9 DATA AVAILABILITY

The replication package [48] contains (i) the questions and code examples used in the experiments, (ii) the questionnaire forms, (iii) detailed results of the qualitative analysis, and (iv) the study results and the R script to analyze them.

ACKNOWLEDGMENTS

Massimiliano Di Penta acknowledges the Italian PRIN 2020 Project EMELIOT "Engineered MachinE Learning-intensive IoT system", ID 2020W3A5FY. Fiorella Zampetti is partially funded by the PON DM 1062/2021 Italian Grant.

REFERENCES

[1] Carol V. Alexandru, José J. Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. 2018. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018*. ACM, 1–11.

[2] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software* 67, 1 (2015), 1–48.

[3] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. 2019. Indentation: simply a matter of style or support for program comprehension?. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 154–164.

[4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300.

[5] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael Maiani de Mello, Anderson Oliveira, Alessandro F. Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. 2020. Atoms of Confusion: The Eyes Do Not Lie. In *34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*. ACM, 243–252.

[6] Robert Dyer and Jigyasa Chauhan. 2022. An exploratory study on the predominant programming paradigms in Python code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 684–695.

[7] Felipe Ebert, Alexander Serebrenik, Christoph Treude, Nicole Novielli, and Fernando Castor. 2022. On recruiting experienced github contributors for interviews and surveys on prolific. In *International Workshop on Recruiting Participants for Empirical Software Engineering*.

[8] Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2020. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering* 25 (2020), 2140–2178.

[9] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 1287–1290.

[10] Github. 2022. Top-Programming-Languages GitHub 2022 https://octoverse.github.com/2022/top-programming-languages. (Last access: 08/11/2023).

[11] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, Sept. 4-8, 2017*. ACM, 129–139.

[12] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the gap from imperative to functional programming through refactoring. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 543–553.

[13] Stefan Hanenberg and Nils Mehlhorn. 2022. Two N-of-1 self-trials on readability differences between anonymous inner classes (AICs) and lambda expressions (LEs) on Java code snippets. *Empirical Software Engineering* 27, 2 (2022), 1–39.

[14] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter identifier names take longer to comprehend. *Empir. Softw. Eng.* 24, 1 (2019), 417–443. https://doi.org/10.1007/S10664-018-9621-X

[15] Prolific Inc. 2023. Prolific https://www.prolific.co. (Last access: 08/11/2023).

[16] Ahmad Jbara and Dror G Feitelson. 2017. How programmers read regular code: a controlled experiment using eye tracking. *Empirical software engineering* 22 (2017), 1440–1477.

[17] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John C. Grundy. 2021. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 432–443. https://doi.org/10.1109/MSR52588.2021.00055

[18] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An Empirical Study Assessing Source Code Readability in Comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 513–523. https://doi.org/10.1109/ICSME.2019.00085

[19] Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability.

[20] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 25–35.

[21] Logilab and Pylint contributors. 2023. Pylint. https://pylint.pycqa.org/

[22] Jeffrey D Long and Norman Cliff. 1997. Confidence intervals for Kendall's tau. *Brit. J. Math. Statist. Psych.* 50, 1 (1997), 31–41.

[23] Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima. 2019. Does the introduction of lambda expressions improve the comprehension of Java programs?. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. 187–196.

[24] Thomas Mccabe. 1996. Cyclomatic complexity and the year 2000. *IEEE Software* 13, 3 (1996), 115–117.

[25] Renée A. McCauley, Brian Hanks, Sue Fitzgerald, and Laurie Murphy. 2015. Recursion vs. Iteration: An Empirical Study of Comprehension Revisited. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE 2015, Kansas City, MO, USA, March 4-7, 2015*. ACM, 350–355.

[26] Nicolas Vandeput. 2022. List Comprehensions vs. For Loops: It Is Not What You Think https://towardsdatascience.com/list-comprehensions-vs-for-loops-it-is-not-what-you-think-34071d4d8207. (Last access: 08/11/2023).

[27] Open AI. 2023. GPTZero https://gptzero.me/. (Last access: 08/11/2023).

[28] Abraham Naftali Oppenheim. 2000. *Questionnaire design, interviewing and attitude measurement*. Bloomsbury Publishing.

[29] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: A Replication Package of an fMRI Study. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 168–169.

[30] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers?: An Eye Tracking Study. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 342–353.

[31] A. Eashaan Rao and Sridhar Chimalakonda. 2020. An Exploratory Study Towards Understanding Lambda Expressions in Python. In *EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020*. ACM, 318–323.

[32] Brittany Reid, Marcelo d'Amorim, Markus Wagner, and Christoph Treude. 2023. NCQ: Code Reuse Support for Node.js Developers. *IEEE Trans. Software Eng.* 49, 5 (2023), 3205–3225.

[33] Brittany Reid, Markus Wagner, Marcelo d'Amorim, and Christoph Treude. 2022. Software Engineering User Study Recruitment on Prolific: An Experience Report. *CoRR* abs/2201.05348 (2022). arXiv:2201.05348

[34] Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, and Mariano Ceccato. 2010. How Developers' Experience and Ability Influence Web Application Comprehension Tasks Supported by UML Stereotypes: A Series of Four Experiments. *IEEE Trans. Software Eng.* 36, 1 (2010), 96–118.

[35] Daniel Russo. 2022. Recruiting software engineers on prolific. (2022).

[36] Giuseppe Scanniello and Michele Risi. 2013. Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 190–199.

[37] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending studies on program comprehension. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*. IEEE Computer Society, 308–311.

[38] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[39] Switowski.com. 2023. For Loop vs. List Comprehension https://switowski.com/blog/for-loop-vs-list-comprehension. (Last access: 08/11/2023).

[40] Mohammad Tahaei and Kami Vaniea. 2022. Lessons Learned From Recruiting Participants With Programming Skills for Empirical Privacy and Security Studies. In *1st International Workshop on Recruiting Participants for Empirical Software Engineering*.

[41] Hiroto Tanaka, Shinsuke Matsumoto, and Shinji Kusumoto. 2019. A study on the current status of functional idioms in Java. *IEICE Transactions on Information and Systems* 102, 12 (2019), 2414–2422.

[42] W. N. Venables and B. D. Ripley. 2002. *Modern Applied Statistics with S* (fourth ed.). Springer, New York. http://www.stats.ox.ac.uk/pub/MASS4 ISBN 0-387-95457-0.

[43] Chak Shun Yu, Christoph Treude, and Maurício Finavaro Aniche. 2019. Comprehending Test Code: An Empirical Study. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 501–512.

[44] Fiorella Zampetti, François Belias, Cyrine Zid, Giuliano Antoniol, and Massimiliano Di Penta. 2022. An Empirical Study on the Fault-Inducing Effect of Functional Constructs in Python. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 47–58.

[45] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. 2022. 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, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 696–708.

[46] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1495–1507.

[47] Mingwei Zheng, Jun Yang, Ming Wen, Hengcheng Zhu, Yepang Liu, and Hai Jin. 2021. Why Do Developers Remove Lambda Expressions in Java?. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 67–78.

[48] Cyrine Zid, Fiorella Zampetti, Giuliano Antoniol, and Massimiliano Di Penta. 2023. *Replication package for the paper: "A Study on the Pythonic Functional Constructs' Understandability"*. https://doi.org/10.5281/zenodo.8191782