Assessing Python Style Guides: An Eye-Tracking Study with Novice Developers

Pablo Roberto
Federal University of Campina Grande
Brazil
pablo@copin.ufcg.edu.br

Rohit Gheyi
Federal University of Campina Grande
Brazil
rohit@dsc.ufcg.edu.br

José Aldo Silva da Costa
State University of Paraíba
Brazil
jose.aldo@servidor.uepb.edu.br

Márcio Ribeiro
Federal University of Alagoas
Brazil
marcio@ic.ufal.br

ABSTRACT

The incorporation and adaptation of style guides play an essential role in software development, influencing code formatting, naming conventions, and structure to enhance readability and simplify maintenance. However, many of these guides often lack empirical studies to validate their recommendations. Previous studies have examined the impact of code styles on developer performance, concluding that some styles have a negative impact on code readability. However, there is a need for more studies that assess other perspectives and the combination of these perspectives on a common basis through experiments. This study aimed to investigate, through eye-tracking, the impact of guidelines in style guides, with a special focus on the PEP8 guide in Python, recognized for its best practices. We conducted a controlled experiment with 32 Python novices, measuring time, the number of attempts, and visual effort through eye-tracking, using fixation duration, fixation count, and regression count for four PEP8 recommendations. Additionally, we conducted interviews to explore the subjects' difficulties and preferences with the programs. The results highlighted that not following the PEP8 Line Break after an Operator guideline increased the eye regression count by 70% in the code snippet where the standard should have been applied. Most subjects preferred the version that adhered to the PEP8 guideline, and some found the left-aligned organization of operators easier to understand. The other evaluated guidelines revealed other interesting nuances, such as the True Comparison, which negatively impacted eye metrics for the PEP8 standard, although subjects preferred the PEP8 suggestion. We recommend practitioners selecting guidelines supported by experimental evaluations.

KEYWORDS

Style Guide, PEP8, Eye tracking camera.

1 INTRODUCTION

Style guides are essential in software development, guiding code formatting, naming conventions, and source code structure to promote readability and facilitate maintenance. Previous studies have investigated the influence of coding styles on readability, with mixed results, indicating that certain styles may compromise code clarity [17]. Major companies, such as Google [20] and Microsoft [23], emphasize style standardization by incorporating guidelines into their corporate style guides, adopting practices aligned with PEP8 to improve code readability and organization.

However, the diversity of programming languages prevents a universal set of style rules, with each language having its own definitions [1]. PEP8, for example, is a widely accepted style guide for Python that suggests best coding practices [14]. Yet, many guides, including PEP8, lack empirical studies as a foundation [31]. Previous studies have examined the impact of coding styles [3] on the performance of novice developers and visual effort, but there is a need for more rigorous assessments that consider the developer's perception and how eye transitions may indicate a greater visual effort with certain style patterns.

Conformity with well-established style practices is crucial to ensuring code quality [32]. PEP8 provides suggestions for coding styles in Python, one of the most popular programming languages. While the PEP8 guide provides justifications for its recommendations, it is crucial to empirically evaluate these recommendations, particularly from dynamic perspectives that consider human factors. For instance, using eye-tracking methodology can provide valuable insights into assessing visual effort [11, 12, 16].

The guidelines outlined in PEP8 not only provide suggestions for improvements but also provide code examples in Python of both incorrect and correct code, as shown in the following listing (Listing 1 and Listing 2) taken from the PEP8 guide. The code in Listings 1 and 2 present the guideline regarding the use of line breaks before or after the operator. In the *PEP8 compliant* version (see Listing 2), the line break should occur before the operator, not after, as in the *PEP8 non-compliant* version (see Listing 1).

Listing 1: *PEP8 non compliant*

```
income = (gross_wages +
          taxable_interest +
          (dividend - quali) -
          ira_deduction -
          stud_interest)
```

Listing 2: *PEP8 compliant*

```
income = (gross_wages
          + taxable_interest
          + (dividend - quali)
          - ira_deduction
          - stud_interest)
```

For the guideline presented in Listing 1, the recommendation states the following: "the eye has to do extra work to figure out which items are added and which are subtracted" [37]. This justification from the style guide led us to question what extra effort the creators of the guide were referring to. In this sense, it becomes important to empirically evaluate the recommendations from the style

guide. In particular, we have to consider eye tracking methodology to assess the eye effort that a particular pattern may generate.

In this study, we employed eye-tracking metrics and conducted interviews with Python novices (32 undergraduate students) to evaluate the impact of four PEP8 guidelines on visual effort and code readability. Our goal was to explore the nuanced relationship between coding patterns and their readability, while also considering the subjective perceptions of developers regarding style preferences and how visual engagement with code might highlight readability issues. Findings suggest that even minor coding patterns recommended by style guides can affect code comprehension, highlighting the need for a more nuanced analysis that includes both objective metrics and the developers' subjective perception.

We evaluate four guidelines (patterns) from the PEP8 style guide: *Whitespace*, *Line Break Before Operator*, *Multiple Statements on the Same Line*, and *Comparison with True*. These guidelines are assessed using a combination of traditional metrics in code comprehension such as time and correct responses, along with eye-tracking metrics including the number of fixations, fixation duration, eye movement regressions, and gaze transitions. Additionally, we conducted interviews with 32 Python novices to understand their preferences and reasons for a particular coding style and correlate them with the results of the assessed metrics.

The obtained results shed light on the potential challenges posed by certain PEP8 guidelines in the context of Python code readability for novices. The findings suggest that adherence to PEP8 guidelines may not always correlate with improved performance, as evidenced by two out of the four evaluated guidelines showing better developer performance in standardized code without PEP8 guidelines. This raises questions about the effectiveness of certain PEP8 recommendations in enhancing code readability, particularly for novice programmers.

This study makes the following contribution:

- An eye tracking controlled experiment with 32 novices in Python to investigate the impact of four guidelines from the PEP8 style guide (Section 4);
- A discussion on the quantitative and qualitative eye-tracking results for the four guidelines from the PEP8 style guide (Section 5).

Additionally, the study suggests the need to revisit and possibly update style guidelines like PEP8 based on empirical data and the practical experiences of developers. By incorporating direct feedback from users and results from studies like this one, guidelines can be refined to better meet the needs of modern developers, balancing code clarity and efficiency with ease of learning and use for new programmers.

2 CODE STYLE AND READABILITY

In Software Engineering, *readability* is related to code clarity, i.e., how easy it is to understand the written expression of the code. Almeida et al. [15] assert that readability is crucial for code maintenance; if the source code is written in a complex manner, the process of understanding the code will require more effort from the reader. This can result in difficulties in identifying bugs, implementing new features, and making modifications. On the other hand, more readable code tends to be easier to modify and debug, making it more sustainable in the long term. Meanwhile, Daka et al. [13] indicate that the visual appearance of code, or style, is generally designated as its readability. The visual organization of code, including formatting, the use of white spaces, and consistency in naming, can significantly affect how developers interpret and interact with the source code. Similarly, Buse and Weimer [6] developed a metric for code readability, demonstrating that certain characteristics of the code can significantly influence how it is understood. This metric considers factors such as the complexity of control structures, clarity in variable and function naming, and code conciseness. Thus, code that follows best practices in readability tends to be more understandable, facilitating the development, maintenance, and collaboration process among team members.

Buse and Weimer [6, 7] and Posnett et al. [26] aimed to identify specific source code characteristics that directly impact its readability and comprehensibility. These features were assessed through the subjective perceptions of students and programmers, offering valuable insights into factors contributing to code's legibility. Furthermore, Lawrie et al. [21, 22] explored code identifiers, examining how different naming styles could affect programmers' ease of understanding. These collective research efforts highlight the importance of deliberate and well-founded coding practices to enhance source code accessibility and maintenance, suggesting that seemingly minor details, like identifier choices, can significantly influence code readability. Our work corroborates findings in this field, focusing on the readability of small code snippets within PEP8 guidelines, with participant subjectivity serving as a key evaluation metric.

3 STUDY DEFINITION

Following the Goal Question Metric approach [2], we aim to analyze Python programs that are PEP8 compliant versus non-compliant for the purpose of comparing them with respect to their impact on code comprehension from the point of view of novices in Python programming language in the context of tasks adapted from introductory programming courses.

We address the following Research Questions (RQs). Our null hypothesis for each RQ is that there is no difference between the *PEP8 compliant* and *PEP8 non-compliant* versions concerning the collected metric.

- **RQ_1: To what extent do the PEP8 guidelines affect the task completion time?** To answer this question, we measured how much time the subject needed to specify the correct output for the task. We also measured how much time the subject spent only in specific areas of the code.
- **RQ_2: To what extent do the PEP8 guidelines affect the number of answer attempts?** To answer this question, we measured the number of attempts the subject made until answering the task correctly, having in mind that the subject is free to make as many attempts as needed.
- **RQ_3: To what extent do the PEP8 guidelines affect the fixations duration and count?** To answer this question, we measured the number and duration of each fixation found in the data captured from the novices. In the code comprehension scenario, fixations with high duration have been associated with an increase in the level of attention [8]. A

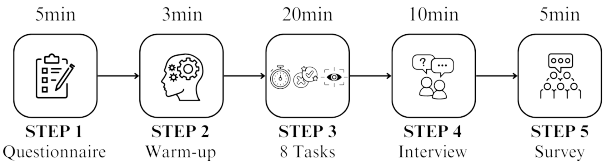

Figure 1: Experiment Steps: Questionnaire, Warm-up, Task, Interview, and Survey.

large number of fixations has been associated with more time to process and understand code statements [4], increased attention to complex code [9], and more visual effort to remember identifier names [34].

- **RQ_4: To what extent do the PEP8 guidelines affect the regressions count?** Just as Rayner [27] observed in reading, regressions can indicate misunderstanding, a concept applied to programming by Busjahn et al. [8]. The study assessed regressions in PEP8 code, measuring backward saccades to quantify readability.

4 METHODOLOGY

The study was structured in five steps (see Figure 1). Initially, we gave the participants a questionnaire to assess their proficiency in Python and introduced them to relevant code examples. The participants were informed about the best posture for eye-tracking data capturing and reminded of their option to leave the study at any time. Following these preparatory steps, we detailed the eye-tracking camera calibration process to the participants, ensuring they were comfortable and correctly positioned for the procedure. This calibration involved participants following on-screen cues to guarantee accurate eye movement tracking, with recalibrations performed as necessary to ensure data reliability.

In the second step, we simulated the execution of the experiment with a simple warm-up task. While they solved the task, we demonstrated how subjects could specify the output, how the subject could close their eyes for two seconds before and after solving the task, how we signaled the correct and incorrect answers, and how we signaled the time limit. The idea is that the subject can feel comfortable with the experiment setup and the equipment.

In the third step, we conducted the actual experiment with eight programs, half adhering to PEP8 guidelines and the other half functionally equivalent but non-compliant with PEP8 guidelines. To avoid learning effects, we used a Latin Square design [5].

In the fourth step, we concluded the experiment with a semi-structured interview. The goal was to obtain qualitative feedback on how subjects examined the programs and their subjective impressions. We went through each of the eight programs and asked three questions: (1) How difficult was it to find the output: very easy, easy, neutral, difficult, or very difficult? (2) Why this perception? (3) How did you find the output?

Finally, we applied a survey in which we presented code excerpts highlighting the use of the PEP8 guideline or not, and we asked the subject's preference, the motivation for the preference, or if they were indifferent to the use of any of those compared excerpts. We were careful with environmental aspects to reduce noise in the data. For example, we did not use a swivel chair because, in previous pilot studies, subjects tended to move, reducing the accuracy of the eye-tracking equipment. Despite the measures we took, obtaining perfect data is virtually impossible, given the limitations of the camera. Therefore, the collected data were processed, analyzed, and interpreted, correcting the data whenever necessary.

4.1 Subjects

Our study included 32 undergraduate students currently pursuing their degrees in the Computer Science field. We considered our subjects as "novices" in Python because they reported having on average seven months of experience in Python, the language in which the programs were written. In general, the subjects had a minimum of six months and a maximum of 42 months of experience with programming languages, including Python, Java, JavaScript, C, and C++. They were recruited from three universities in Brazil, mainly through in-person invitations or text messages. All subjects were Brazilian Portuguese speakers enrolled in academic institutions.

Regarding the sample size, we performed a calculation considering the desired effect, significance level, and statistical power. The goal was to ensure a minimum power of 0.8, with a significance level of 0.05, using the t-test sample size calculation. Our analysis indicated that 26 subjects in two groups would be required to meet these criteria. Alternatively, given that we had 32 subjects instead of 26, our study can identify a moderate effect size of 0.5, maintaining a statistical power of 0.8 and a significance level of 0.05. A larger sample size provides sufficient sensitivity to detect a slightly smaller effect while maintaining statistical robustness.

We conducted the experiment at three locations to gather more subjects and to have a variety of subjects from different higher education institutions. However, different locations may influence subjects' visual attention. To mitigate this, we carefully organized the rooms to have similar conditions. For instance, the rooms were quiet, with minimal distractions, similar temperatures, and artificial light sources. We documented which subject performed the experiment at each location to account for potential differences.

4.2 Design

As illustrated in Figure 2, each subject analyzed eight programs (P_1-P_8). To mitigate learning effects, we employed the Latin Square design [5]. Sixteen different programs were designed, divided into two sets of programs (SP_1 and SP_2). A subject analyzed four programs from set SP_1 and four programs from set SP_2. Another subject analyzed four programs from set SP_1 and four programs from set SP_2. Programs within the same set, although having different code programs, resulted in the same output. In all programs, subjects were required to specify the correct output, with no multiple-choice options. Given the program's input, subjects had to perform tasks such as calculating operations, and summing lists, among others.

4.3 Evaluated PEP8 Style Guide Guidelines

We evaluated four guidelines from the PEP8 style guide (see Table 1): *Whitespace*, *Line Break Before Operator*, *Multiple Statements on the Same Line*, and *Comparison to True*. Following the guide's suggestions, we referred to the snippets as *PEP8 compliant* and

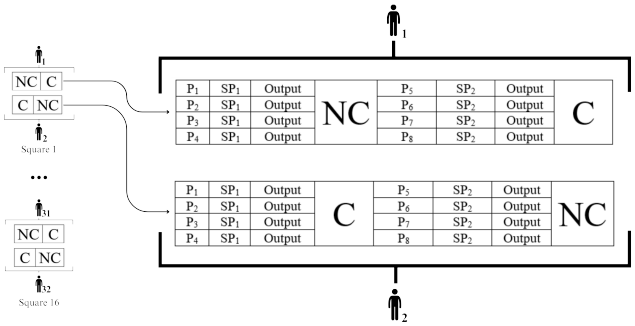

Figure 2: Latin Square Structure. Each subject received four programs (P_1-P_4), which were *PEP8 compliant* (C). These programs belonged to Program Set 1 (SP_1). Additionally, the subject received four programs (P_5-P_8) from Program Set 2 (SP_2), comprising *PEP8 non-compliant* (NC).

PEP8 non-compliant. We selected PEP8 guidelines suitable for code snippets for novices in Python.

Table 1: PEP8 guidelines evaluated in this study.

Guideline	PEP8 compliant	PEP8 non-compliant
White-space	`hypot2 = x*x + y*y` `c = (a+b) * (a-b)`	`hypot2 = x * x + y * y` `c = (a + b) * (a - b)`

Line Break Before Operator	`income = (gross_wages`
	`+ taxable`
	`+ dividends`
	`- ira_deduction`
	`- student)`
		`income = (gross_wages +`
		`taxable +`
		`dividends -`
		`ira_deduction -`
		`student)`

Multiple Statements on the Same Line	`while t < 10:` `t = delay()`	`while t < 10: t = delay()`
Comparison to True	`if greeting:`	`if greeting == True:`

The selected code snippets and guidelines for analysis were carefully chosen with a focus on the target demographic's background knowledge, the novice Python students. We have considered that Python served as the primary language covered in the Algorithms or Structured Programming courses at the universities where our experiment was conducted. Our objective was to explore and evaluate guidelines that align with the prevalent coding styles adopted by novice Python learners in these academic settings.

4.4 Programs

We selected code snippets from repositories such as GeekForGeeks[1] and Leetcode[2], for introductory programming activities. We prioritized problems with up to 11 lines of code, adapted for camera constraints, as illustrated in Figure 3. Following a common methodology employed by code comprehension experiments [25] [18] [33],

[1] https://www.geeksforgeeks.org/
[2] https://leetcode.com/

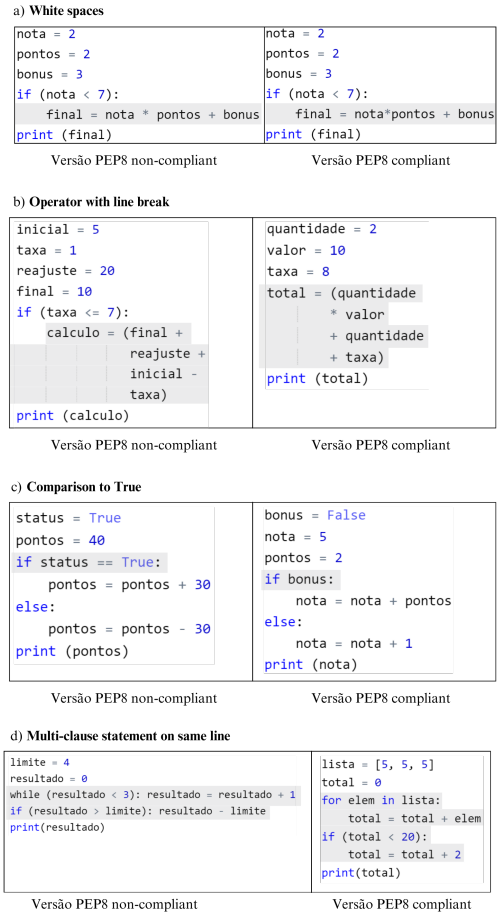

Figure 3: Programs presented to the participants.

we asked subjects to predict the correct output of code snippets, without syntactic errors, addressing the four PEP8 guidelines evaluated in this study (Table 1).

The programs, displayed in Consolas 16, underwent a careful approach. Each program, whether the *PEP8 compliant* or *PEP8 non-compliant* version, had a single instance of one of the PEP8 style guide patterns. With a Latin Square design, no subject saw both the *PEP8 compliant* or *PEP8 non-compliant* versions of the same program.

In Figure 3, we present examples of the programs used in the experiment, with one version containing the *PEP8 non-compliant* pattern and another version containing the *PEP8 compliant* pattern of PEP8 for the four styles evaluated. In Figure 3, we depict a set of programs with *PEP8 non-compliant* versions (left-hand side) of the code, including guidelines such as Whitespace, Line Break Before Operator, Comparison to True, and Multiple Statements on the Same Line. The shaded areas indicate Areas of Interest (AOIs), corresponding to lines of code where the *PEP8 non-compliant* and *PEP8 compliant* versions differ. We chose to use small-sized programs,

with up to 11 lines of code, to fit the code on the screen. This choice may limit the applicability to more extensive programs.

4.5 Eye Tracking System

Our research employed the Tobii Eye Tracker 4C with a sampling rate of 90 Hz. The eye tracker calibration followed the standard procedure, involving gazes at five calibration points, twice, and verification with eight points. The device was mounted at a distance of 50-60 cm from the subject on a laptop screen. Code tasks were displayed in full-screen mode, without the use of an Integrated Development Environment (IDE). We calculated a precision error of 0.7 degrees from this distance. For eye gaze analysis and metric collection, we developed a Python script. We used camera settings used in previous studies [11, 12] on code comprehension using eye-tracking cameras We used a Dispersion-Based algorithm to classify the fixations. In particular, we used the Dispersion-Threshold Identification [30]. We also classified gaze samples as belonging to a fixation if the samples are located within a spatial region of approximately 0.5 degrees [24]. We also implemented a simple Python script to create diagrams from data points using open source libraries to draw arrows and images, and create heatmaps

4.6 Study Pilot

We conducted pilot studies with four participants to refine materials and adjust the experiment's design, excluding these subjects from the final analysis. The process allowed us to simplify the programming tasks and focus exclusively on the impact of PEP8 guidelines on the codes, identifying and eliminating other variables that could influence the results.

The study material included a collection of programs, a questionnaire for characterizing participants, and semi-structured interview questions. Program snippets were sourced from the PEP8 guide and introductory programming course datasets. Various aspects like code difficulty, font size, style, spacing, and indentation were evaluated. Tasks generally took under two minutes to complete, and questionnaire questions were refined. Identifiers were carefully chosen to convey information, such as using abbreviations like `elem` and specific terms like `bonus` for context clarity.

5 RESULTS AND DISCUSSION

In Table 2, we summarize the quantitative results of the collected metrics for each guideline with the statistical analysis. We present two perspectives of the metrics evaluated in this work, one examining only the AOI and the other examining the code as a whole. While time in the code, for example, consists of the time needed to examine and solve the task, regardless of the fixations made, time in the AOI consists of the time dedicated to examining only the region containing a style following or not following the PEP8 guidelines.

Table 2 also presents the data for the *PEP8 non-compliant* version (column NC) and *PEP8 compliant* version (column C) on the metrics highlighted in the second column. In the PD% column, we present the percentage difference between the two versions concerning the particular metric. The percentage was calculated with respect to the NC version with an arrow that indicates how much the C version increased or decreased this percentage compared to the NC version. The NC and C columns are based on the median as a measure of central tendency, except for less sensitive attempts which are based on the mean. While time in the code represents the total effort to examine and solve the task, time in the AOI offers more specific insights, focusing exclusively on the region relevant to the PEP8 guideline.

Concerning our RQ_1, the *PEP8 compliant* version of the PEP8 guideline *Line Break Before Operator* resulted in a reduction by 48.7% in time in the AOI and 35.27% in total time compared to the NC version. This suggests that following the PEP8 guideline can optimize the time spent on code analysis. It also highlights the importance of considering not only the total duration but also the efficiency in the code analysis. Notably, this distinction between time in the code and time in the AOI is vital when evaluating the NC and C versions, as it provides a more refined perspective on how following the PEP8 guidelines can influence not only the total time invested but also the efficiency and accuracy in the code analysis. This difference between the NC and C versions can be critical for understanding the overall impact of coding practices according to the established guidelines.

By following the *Line Break Before Operator* guideline, Table 2 highlights a decrease by 37.14% in Fixation Duration, our RQ_3. This result points to a reduction in visual effort concentrated in the area associated with the evaluated pattern when PEP8 guidelines are followed compared to the NC version. Additionally, when analyzing the metrics of Horizontal Regressions and Vertical Regressions for the *White Space* guideline in our RQ_4, we observe that following the PEP8 guidelines is associated with a 25% reduction in Horizontal Regressions, indicating a smoother and continuous reading.

Concerning the *Comparison to True* guideline, Table 2 presents important nuances regarding the influence of PEP8 guidelines on various metrics. Although following this guideline resulted in an 8.94% reduction in time in the AOI, it showed an increase of 16.84% in time in the code (RQ_1). There is also a simultaneous increase of 16.66% in Fixation Duration and 8.25% in Fixation Count (RQ_3). These results suggest a potential trade-off between overall efficiency and detail in the code analysis, indicating that, while following the PEP8 guideline may speed up the analysis, there may be a cost associated with the frequency and duration of revisions.

Additionally, we observed that in general, there were no significant differences in time spent and the number of answer attempts between the different PEP8 patterns for the *White Space Multiple Statements on the same Line* and *Comparison to True* criteria. However, there was a noteworthy decrease in fixation duration and count for the *Line Break Before Operator* and *Multiple Statements on the same Line* patterns, indicating enhanced efficiency in understanding and processing these patterns. Moreover, a decrease in the number of horizontal and vertical regressions was observed for these same patterns, suggesting a more linear and organized reading of the code. Interestingly, certain metrics, such as time and fixation count, exhibited performance differences between the PEP8 patterns compared to the control group *Comparison to True*, suggesting variations in the ease of understanding and processing the different patterns. Next we discuss our results in more details.

Table 2: Summary of metrics: time, submissions, fixation duration, fixation count, horizontal regressions, and vertical regressions for each PEP8 guideline. NC represents the *PEP8 non-compliant* version and C the *PEP8 compliant* version; PD represents the percentage difference; n/a represents unassigned value. Bold font represents statistically significant differences.

PEP8 Guidel.	Metrics	In the AOI				In the Code			
		NC	C	PD%	*p*-val.	NC	C	PD%	*p*-val.
White Space	Time	6	5.69	↓5.25	0.82	20	18.91	↓5.45	0.63
	Submissions	n/a	n/a	n/a	n/a	1	1	0	0.36
	Fixations Duration	10	9.5	↓5	0.86	23.5	27	↑12.7	0.97
	Fixations Count	2.92	2.96	↑1.54	0.7	7.17	8.59	↑19.73	0.91
	Horizontal Regressions	2	1.5	↓25	0.46	4	4	0.86	0.86
	Vertical Regressions	0	0	n/a	n/a	6	7	↑8.33	0.96
Line Break Before Operator	Time	14.27	7.32	↓48.7	**0.002**	30.7	19.8	↓35.27	**0.005**
	Submissions	n/a	n/a	n/a	0.0	1.13	1	↓11.11	**0.04**
	Fixations Duration	17.5	11	↓37.14	**0.02**	36	31.5	↓12.5	0.11
	Fixations Count	5.46	4.21	↓22.8	**0.01**	11.06	10.11	↓8.63	**0.04**
	Horizontal Regressions	6	2	↓66.66	**0.03**	6	4	↓27.27	**0.02**
	Vertical Regressions	2.5	1	↓60	**0.03**	12	85	↓29.16	**0.007**
Multiple Statements on the same Line	Time	28.95	22.89	↓20.92	0.2	42.9	32.5	↓24.16	0.06
	Submissions	n/a	n/a	n/a	n/a	1.5	1.13	↓25	0.1
	Fixations Duration	46.5	32.5	↓30.1	0.36	62	45	↓27.41	0.21
	Fixations Count	14.99	11.58	↓27.74	0.37	20.04	15.58	↓22.24	0.15
	Horizontal Regressions	13	14	↑7.69	0.76	13	10	↓24	0.33
	Vertical Regressions	2.5	4	↑60	0.63	11	9.5	↓13.63	0.8
Comparison to True	Time	2.46	2.24	↓8.94	0.87	13.6	15.9	↑16.84	0.05
	Submissions	n/a	n/a	n/a	n/a	1	1	0	n/a
	Fixations Duration	3	3.5	↑16.66	0.56	18	22	↑22.22	0.28
	Fixations Count	1.11	1.03	↑8.25	0.31	7.02	6.21	↑13.12	0.16
	Horizontal Regressions	0.5	0	0	0.05	2	3	↑25	0.16
	Vertical Regressions	0	0	n/a	n/a	5	6	↑20	0.15

5.1 Preferences of the Subjects

After solving all programming tasks, we showed the *PEP8 compliant* and *PEP8 non-compliant* versions, side by side, for each PEP8 guideline, to the subjects. The differences in the code snippets between the *PEP8 compliant* and *PEP8 non-compliant* versions were highlighted. Figure 4 depicts the subjects' overall responses, indicating their preferences among the presented versions, expressing whether they had a Strong Preference, Preference, or were Indifferent to the versions presented: Strongly Prefers (SP) *PEP8 non-compliant* (SPNC) or *PEP8 compliant* (SPC), Prefers (P) *PEP8 non-compliant* (PNC) or *PEP8 compliant* (PC), and Indifferent (I).

Subjects predominantly preferred or strongly preferred the *PEP8 non-compliant* version for the *White Spaces* guideline. However, concerning the guideline for *Multiple Statements on the Same Line*, the majority preferred or strongly preferred the *PEP8 compliant* version. For *Line Break Before Operator* and *Comparison to True*, the majority preferred the *PEP8 compliant* version with PEP8 guidelines. This response pattern may suggest that while some PEP8 guidelines are widely accepted, others are considered less critical or subject to personal interpretation.

In addition to expressing preferences among the presented versions, the subjects were asked about the reasons behind their choices. The more in-depth analysis of these interview data is enriched through triangulation with eye tracking metrics. The discussion of this triangulation is presented in the subsequent sections as we investigate each pattern of the PEP8 guidelines.

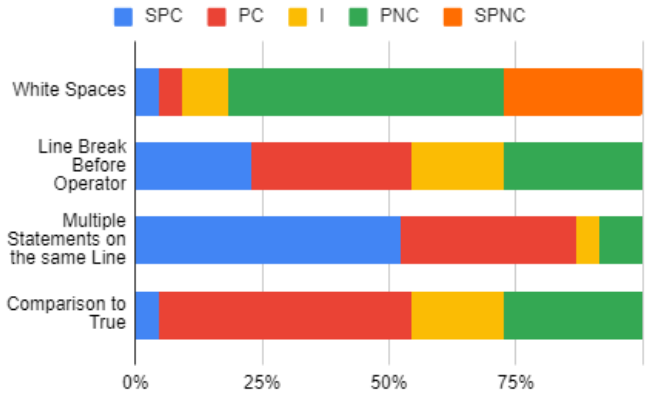

Figure 4: Subject's preferences for the *PEP8 compliant* and *PEP8 non-compliant* versions of the PEP8 guidelines. We used the following acronyms: Strongly Prefers *PEP8 compliant* (SPC); Prefers *PEP8 compliant* (PC); Indifferent (I); Prefers *PEP8 non-compliant* (PNC); Strongly Prefers *PEP8 non-compliant* (SPNC).

5.2 PEP8 Guidelines

In this section, we triangulate and discuss each guideline evaluated, starting with *White Space* (see Section 5.2.1), followed by *Line Break Before Operator* (see Section 5.2.2), *Multiple Clauses* (see Section 5.2.3), and finally, *Comparison to True* (see Section 5.2.4).

5.2.1 White Space. Regarding our RQ_1, it was observed that subjects spent more time in the AOI in the *PEP8 non-compliant*. Apparently, adding space between the multiplication operator also caused subjects to regress more in the *PEP8 non-compliant* version. Not following the PEP8 guideline impacted the number of horizontal eye regressions for the novices, reducing it by approximately 25%. To clarify this aspect further, we will discuss how eye regression can indicate issues in code readability with the *PEP8 non-compliant* version of the PEP8 recommendation for *White Space*.

Still concerning our RQ_4, it was noted that the *PEP8 non-compliant* version exhibited a slightly higher number of horizontal regressions in the AOI. From data extracted from one of the subjects in the experiment, Figure 5 presents the horizontal regressions captured by this subject's eye tracking camera. On the left-hand side of Figure 5, we have the code snippet containing the *PEP8 non-compliant* version of PEP8 guideline *White Space* (Table 2), and on the right-hand side, the version with the *PEP8 compliant* guideline. Each line between code blocks indicates either a code *regression*, where the gaze returns to a previous section of code, or *progression*, where the gaze moves on to a further section. The spacing around the multiplication operator (seen on the left-hand side of Figure 5) may account for an increase in horizontal regressions, potentially indicating a greater visual effort and reading repetition.

Figure 6 showcases two code excerpts marked with gaze transitions from different subjects, one with the *PEP8 non-compliant* version (a) and the other with the *PEP8 compliant* version (b), according to PEP8's *White Space* standard. The color-coded lines track the sequence of eye movements across the code versions. It is observed that the *PEP8 compliant* version (b) had fewer regressions, suggesting that code adhering to the PEP8 *White Space* guideline is easier to follow and understand, as opposed to the *PEP8 non-compliant* version (a), where more frequent regressions occurred, especially in the section not following the PEP8 *White Space* guideline, a pattern repeated across subjects.

Regarding the subjects' preferences, approximately 81% of the subjects preferred the *PEP8 non-compliant* version of PEP8 guide for *White Space*. When we asked why they preferred one style over the other, one of the subjects responded as follows: "...I don't like the idea of [the operator followed by the variable without using space], I think the lack of spacing is confusing...". The absence of space between the multiplication operator displeases some subjects, which may justify the preference for the *PEP8 non-compliant* pattern. However, we can conclude that although the majority of subjects preferred the *PEP8 non-compliant* pattern, it negatively impacted code readability when considering overall time and eye regression data. Interestingly, participants' comments on specific tasks indicate that, despite a preference for more spaces, understanding of the code was not necessarily improved by this practice.

5.2.2 Line Break Before Operator. For the *Line Break Before Operator* guideline, the PEP8 guide provides the following justification:

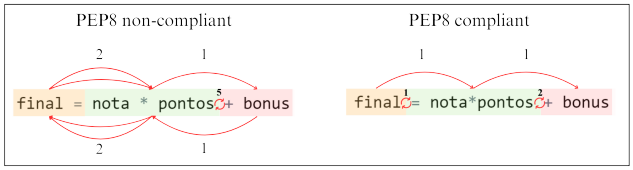

Figure 5: Eye *regression* and *progression* of reading in the AOI of the *White Space* pattern for the *PEP8 non-compliant* version (left-hand side) and *PEP8 compliant* version (right-hand side) of a subject. The green region contains the portion of the code where the PEP8 pattern has been applied or not. Arrows pointing from right to left represent reading progression, and from left to right, eye returning to the previous region of the code. The numbers on the arrows represent the number of times progression or regression occurred.

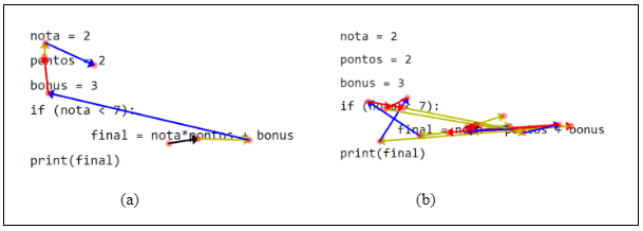

Figure 6: Sequential gaze transitions when reading a program with the *White Space* pattern for the *PEP8 non-compliant* (a) and *PEP8 compliant* (b) versions of different subjects.

"the eye has to do extra work to figure out which items are added and which are subtracted" [37]. In our study, we found that the *PEP8 non-compliant* version reduced the performance of the subjects in nearly all evaluated metrics, as shown in Table 2, corroborating the findings of Rossum et al. [37]. Statistically, the version recommended by PEP8 for *Line Broke Before Operator* demonstrated better performance. Regarding our RQ_4, there is indeed an indication of more vertical and horizontal regressions for the *PEP8 non-compliant* guideline, supporting PEP8's assertion.

In Figure 7(a) and (b), we illustrate the eye transition where two subjects regress their gaze to the operator at a certain point in the program reading, suggesting an extra visual effort to return to the operator (PEP8 non-compliant version). This behavior was also observed in the data from other subjects. However, in the snippet from Figure 7(c) with the *PEP8 compliant* version, one subject demonstrates a sequential reading flow without regressing, meaning they do not go back to the operator. This observation was noted in the data from other subjects for the *PEP8 compliant* version of the pattern discussed in this subsection.

For the *Line Break Before Operator* guideline, the majority prefers the *PEP8 compliant* version. The main reasons were "left alignment" and "ease of reading", stating that they are more accustomed to a left-aligned format for mathematical operations. This observation complements the discussion on the preference for line breaks before or after a binary operator, which revealed divided opinions among

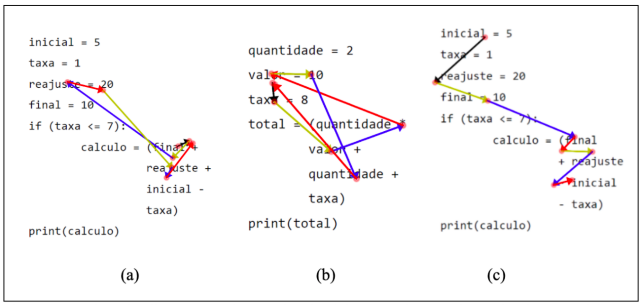

Figure 7: Eye transition for *Line Break Before Operator* guideline from three subjects, with code snippets (a) and (b) representing the *PEP8 non-compliant* version of the guideline and code (c) representing the *PEP8 compliant* version.

participants. Some found that breaking lines facilitated understanding by visually separating the components of complex calculations, despite not being accustomed to this formatting. Others, however, expressed that this approach caused estrangement and preferred the presentation of calculations in a single line, arguing that this made the sequence of operations more straightforward and easier to follow. This divergence of opinions highlights how personal familiarity significantly influences the perception of code readability, with some valuing the clarity provided by line breaks in extensive operations, while others see it as a barrier to the immediate understanding of mathematical operations.

5.2.3 Multiple Clauses on the Same Line. Answering our RQ_2, the number of submissions for the *PEP8 non-compliant* version was slightly higher than for the *PEP8 compliant* one regarding the guideline *Multiple Statements on the Same Line*. To better understand it, we analyze one of the subjects' comments: "...separating, we understand better what the if does because we can get confused if the `if` is inside the `while`...". In this comment, it is possible to identify that the lack of indentation can cause confusion. According to this subject, there is confusion when one clause is followed by another on the same line.

Concerning the fixations count, our RQ_3, we observed that the *PEP8 compliant* version reduced the number of fixations by up to 30% and the duration of these fixations by up to 27% in the AOI for the guideline discussed in this subsection, compared to the *PEP8 non-compliant* pattern. From the heatmap and fixation count, it was possible to notice that, for two subjects, we were able to identify some nuances that may justify these data (Figure 8).

The right-hand side (a) in Figure 8 demonstrates the code with the *PEP8 compliant* version. Both code snippets on side (a) and on side (b) have the same algorithmic complexity. However, we noticed that the performance of the subject who solved the *PEP8 non-compliant* version was worse in terms of time, our RQ_1, number of fixations and fixation duration, our RQ_3, in the AOI. The overall data presented in Table 2 also reflects this difference.

Regarding preference, Figure 4 shows that almost all subjects prefer the *PEP8 compliant* version of PEP8 for the guideline *Multiple Clauses in the Same Line*. The reason reported by some Python novices was that in the *PEP8 compliant* version, there is more clarity about whether the `if` statement is inside the `while` loop or not.

The strong preference for compliance with PEP8 underscores the importance of readability and clarity in code, reflecting familiarity with coding practices that prioritize these aspects. Participants found tasks easier due to the clear structure of the code, as recommended by PEP8, which facilitates comprehension even with multiple variables and operations. These well-defined and transparent standards improve code readability and assist beginners in learning programming.

5.2.4 Comparison to True. In Figure 9(a), we observed that a subject analyzed the `else` block in the code, which would not be an expected behavior, given that the value of `status` is `True`. In Figure 9(b), it can be noted that the subject returned several times to the `if` block, probably to check the value of `status`. One of the subjects even mentioned that the value of `status` without the comparison to `True` is not clear. This comment justifies the gaze behavior in the section related to *status*.

Regarding the regression analyses (RQ_4), Figure 10 depicts two graphs where the node represents a line of code (left and right-hand sides), and the edges indicate regressions, progressions, or returns to the same line of code. The *PEP8 non-compliant* pattern shows a lower number of gaze returns compared to the *PEP8 compliant* pattern. In fact, after checking the value of `bonus` in the `if` block (which is `False`), both in the *PEP8 non-compliant* pattern, as represented in Figure 10(a), and in Figure 9(c), subjects direct their gaze to the `else` block in the code. This behavior is expected considering the value of `bonus`. However, this observation does not apply to the *PEP8 compliant* pattern in Figure 10(b), as subjects explore code snippets that do not require verification. Furthermore, when comparing eye tracking data for the code snippets in Figure 9(b) and (d), the *PEP8 non-compliant* version reveals that the subject's reading was more sequential and did not involve returns to check the value of `status`, as observed in the *PEP8 compliant* pattern.

The *PEP8 non-compliant* showed better performance, contradicting PEP8 guidelines. Participant feedback suggests that direct comparison of boolean values with `True` would be more appropriate, indicating performance improvement. This method simplifies boolean conditions, promoting readability and quick decision-making, especially useful for novices. PEP8 could revisit coding style guidelines.

6 THREATS TO VALIDITY

We conducted the experiment at three locations to gather more subjects and to have a variety of subjects from different higher education institutions. However, different locations may influence subjects' visual attention. To mitigate this, we carefully organized the rooms to have similar conditions. For instance, the rooms were quiet, with minimal distractions, similar temperatures, and artificial light sources. We documented which subject performed the experiment at each location to account for potential differences.

We allocated a total time of 40 minutes for each subject and assigned them eight programs, which could have influenced visual effort. To minimize this threat, we designed simple and short programs, each with only one instantiated pattern. Given the simplicity of the programs, most subjects solved them before the time limit.

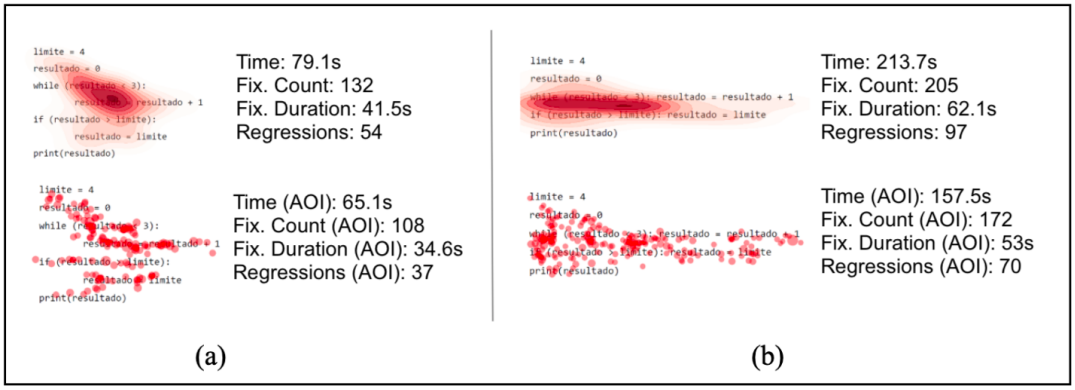

Figure 8: Heatmaps and Fixation for the *Multiple Clauses on the Same Line* pattern.

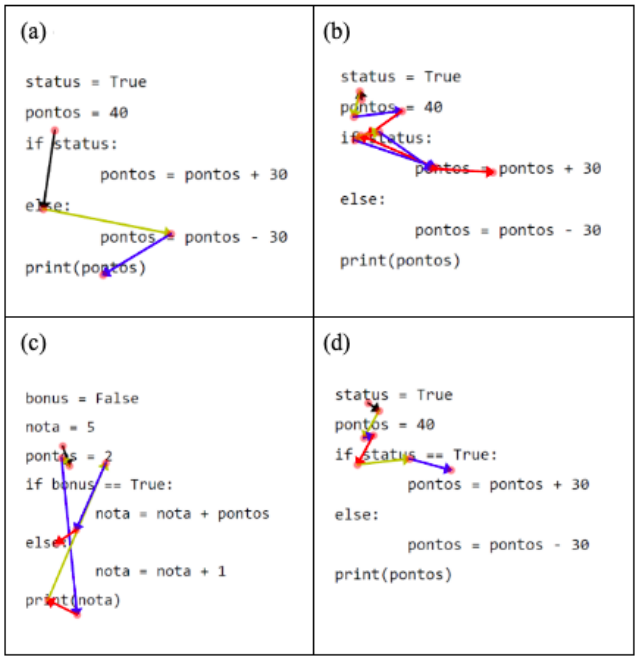

Figure 9: Eye tracking transition for the *Comparison to True* guideline. Versions (a) and (b) are *PEP8 compliant*, while versions (c) and (d) are *PEP8 non-compliant*, according to PEP8, representing four distinct subjects.

We chose to use small-sized programs, with up to 11 lines of code, to fit the code on the screen. This choice may limit the applicability to more extensive programs. However, previous works have employed code snippets with a similar number of lines, such as in Costa et al. [11]. If we identify disparities in short snippets, we expect that longer segments may reveal more pronounced differences. Nevertheless, to support such expectations, it is imperative to conduct further studies with more extensive code snippets.

Within the scope of our study, we directed our attention to Python novices, limiting the generalization to more experienced developers in the language. However, using students as participants remains a valid simplification of reality needed in laboratory contexts [19, 29]. We plan to explore the same topic of this study with more experienced developers in the future.

7 RELATED WORK

Previous approaches [3, 17] identified that certain coding styles negatively affect Java code readability, with whitespace being a particular concern. Our eye-tracking study revealed that in Python, following PEP8's *White Space* guidelines may not align with better readability, as less spacing could potentially reduce visual regression and improve comprehension.

Sharafi et al. [35] examined the influence of Camel Case and underscore coding styles on code comprehension, measuring time, response correctness, and visual effort through eye tracking. They found a significant improvement in the time and visual effort with the underscore style. In a subsequent investigation [34] on the same styles, considering the gender of the subjects, no significant differences were identified in metrics such as time, accuracy, and visual effort. In our study, we sought similar metrics, namely time, number of attempts, and visual effort, however, in a different context.

Stefik and Siebert [36] investigated how programming language syntax affects the comprehension of novice programmers. They asked novices to assess the intuitiveness of different programming language constructs and found that syntactic choices in commercial programming languages tend to be more intuitive for novices, influencing their initial programming accuracy rates. In contrast, our approach also considered the context of novices but specifically focused on a programming language. Additionally, we employed objective metrics to evaluate the performance of novices during code-solving tasks, providing a structured and quantitative analysis of code readability and how it could impact the accuracy of responses to proposed tasks containing the patterns.

Costa et al. [10, 11] used eye tracking to study how code patterns such as *atoms of confusion* affect the understanding of Python code among novices. Their experiment with 32 participants showed that these atoms can complicate code comprehension, leading to increased effort and more attempts to solve coding problems. This research underscores the use of eye tracking in identifying which code patterns most hinder the novices' code comprehension.

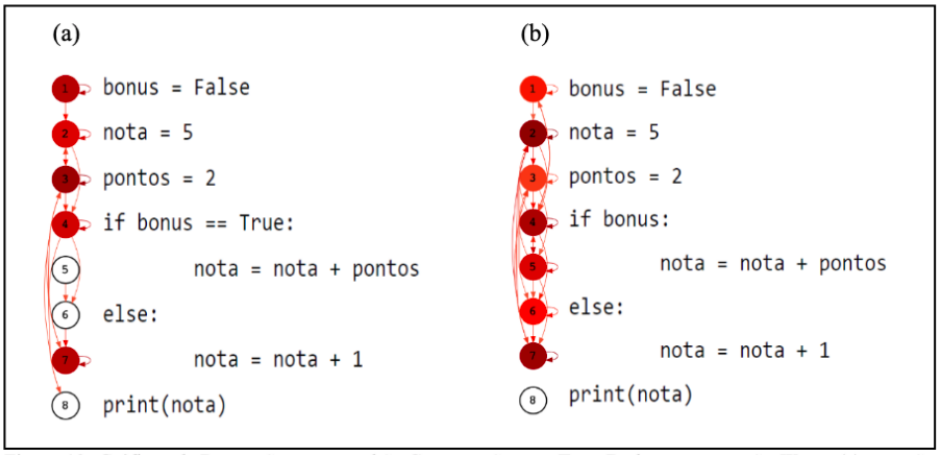

Figure 10: Regression Graphs for the *Comparison to True* guideline. Data for the *PEP8 non-compliant* version (a) and *PEP8 compliant* version (b) from two distinct subjects.

Sharif et al. [28] present a guideline to conduct eye tracking studies. We followed rigorous guidelines for eye-tracking studies to examine how minor coding styles, as recommended by PEP8, affect Python code readability. Our findings reveal differences in readability and stylistic preferences, showcasing eye tracking's capability to measure the influence of coding patterns on both the performance and perception of programmers, thus highlighting its critical role in enhancing the understanding of coding practices and guiding future research.

8 CONCLUSIONS

In this work, we explore eye tracking as a method to gain insights into the guidelines of a code style guide. We conducted a controlled experiment with an eye tracker to assess the impact of four PEP8 guidelines on code readability, analyzing how the *PEP8 compliant* and *PEP8 non-compliant* versions, as defined by PEP8, affected the time, number of attempts, and visual effort of 32 Python novices.

We observed that non-adherence to PEP8's proper spacing increases time, number of fixations and horizontal regressions. Moreover, omitting line breaks before operators, against PEP8 recommendations, increased regressions, validating the importance of these practices for code readability. The analysis of the *Multiple Clauses on the Same Line* pattern showed that following the PEP8 guideline to separate clauses onto different lines reduced both the number and duration of fixations, enhancing code comprehension due to the clarity provided by this separation. Surprisingly, for the *Comparison to True* pattern, results suggested that direct comparisons with Boolean values (True/False) were more effective, indicating that, in certain cases, deviating from PEP8 recommendations might actually aid in novice programmers' understanding of the code.

For educators, we recommend paying close attention to the guidelines used in classes that impact undergraduate students' program comprehension. Based on our findings, explicitly using `== True` in conditions may help novices better understand the code. For practitioners, it is crucial to thoroughly understand coding style guidelines before adopting them. We recommend selecting guidelines supported by experimental evaluations rather than persuasive statements. For researchers, we recommend evaluating more coding styles using robust methodologies to understand the impact of each style on code comprehension. It is also important to propose coding styles like PEP8 using proper methodologies. The use of eye-tracking cameras can help researchers identify gaze transition patterns that could be integrated into more advanced IDEs. Advanced IDEs could use eye-tracking cameras to monitor developers' gaze transitions. These cameras, equipped with machine learning models and patterns of gaze transitions, could help automatically adjust code based on developers' preferences.

For future research, we will explore the effects of lesser-studied PEP8 guidelines and their differing impacts on novice versus experienced developers. Longitudinal studies to track the evolution of guideline comprehension over time, assessing how guideline adherence influences code maintainability, and extending the analysis to style guides across various programming languages are also recommended to enhance both educational strategies and software development practices. We intend to investigate more extensive code and the impact of guidelines on gender and neurodiversity. We aim at analyzing the *Comparison to True* guideline further by interviewing more undergraduate students to better understand the results.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions. This work was partially supported by CNPq and FAPEAL grants.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'14)*. 281–293.
[2] Victor Basili, G. Caldiera, and H. Rombach. 1994. The Goal Question Metric Approach. *Encyclopedia of Software Engineering* (1994), 528–532.

[3] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. 2019. Indentation: Aimply a Matter of Style or Support for Program Comprehension?. In *Proceedings of the International Conference on Program Comprehension (ICPC'19)*. IEEE, 154–164.

[4] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.

[5] George Box, J. Stuart Hunter, and William G. Hunter. 2005. *Statistics for Experimenters*. Wiley-Interscience.

[6] Raymond Buse and Westley Weimer. 2009. Learning a Metric for Code Readability. In *Proceedings of the International Symposium on Software Testing and Analysis*. 465–475.

[7] Raymond P. L. Buse and Westley R. Weimer. 2008. A Metric for Software Readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM Press, 121–130.

[8] Teresa Busjahn, Carsten Schulte, Sascha Tamm, and Roman Bednarik. 2015. Eye Movements in Programming Education II: Analyzing the Novice's Gaze. In *Proceedings of the Conference on Computing Education (ICER'15)*.

[9] Martha Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The Roles Beacons Play in Comprehension for Novice and Expert Programmers.. In *Workshop of the Psychology of Programming Interest Group (PPIG'02)*. 5.

[10] José Aldo Silva da Costa and Rohit Gheyi. 2023. Evaluating the Code Comprehension of Novices with Eye Tracking. In *Concurso de Teses e Dissertações em Engenharia de Software (CTD-ES)*.

[11] José Aldo Silva da Costa, Rohit Gheyi, Fernando Castor, Pablo Roberto Fernandes de Oliveira, Márcio Ribeiro, and Baldoino Fonseca. 2023. Seeing Confusion through a New Lens: on the Impact of Atoms of Confusion on Novices' Code Comprehension. *Empirical Software Engineering* 28, 4 (2023), 81.

[12] José Aldo Silva da Costa, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Vander Alves, Baldoino Fonseca, Flávio Medeiros, and Alessandro Garcia. 2021. Evaluating Refactorings for Disciplining #ifdef Annotations: An Eye Tracking Study with Novices. *Empirical Software Engineering* 26, 5 (2021), 1–35.

[13] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the Foundations of Software Engineering*. 107–118.

[14] Subhasish Dasgupta and Sara Hooshangi. 2017. Code Quality: Examining the Efficacy of Automated Tools. In *Americas Conference on Information Systems (AMCIS'17)*.

[15] Jorgy Rady de Almeida, João Batista Camargo, Bruno Abrantes Basseto, and Sérgio Miranda Paz. 2003. Best Practices in Code Inspection for Safety-critical Software. *IEEE Software* 20, 3 (2003), 56–63.

[16] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. 2020. Atoms of Confusion: The Eyes Do Not Lie. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES'20)*. 243–252.

[17] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of Coding Practices on Readability. In *Proceedings of the International Conference on Program Comprehension (ICPC'18)*. 277–285.

[18] 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.

[19] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering* 23, 1 (2018), 452–489.

[20] Google. 2024. *Google Python Style Guide*. https://google.github.io/styleguide/pyguide.html

[21] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. 2006. What's in a name? A study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 3–12.

[22] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering* 3 (2007), 303–318.

[23] Microsoft. 2024. *Formatting Python Code*. https://learn.microsoft.com/en-us/visualstudio/python/formatting-python-code?view=vs-2022

[24] Marcus Nyström and Kenneth Holmqvist. 2010. An adaptive algorithm for fixation, saccade, and glissade detection in eyetracking data. *Behavior research methods* 42, 1 (2010), 188–204.

[25] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'20)*. 348–359.

[26] D. Posnett, A. Hindle, and P. Devanbu. 2011. A Simpler Model of Software Readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*. ACM Press, 73–82.

[27] Keith Rayner. 1998. Eye Movements in Reading and Information Processing: 20 Years of Research. *Psychological Bulletin* 124, 3 (1998), 372.

[28] Sharafi Zohreh; Bonita Sharif; Yann-Gaël Guéhéneuc; Andrew Begel; Bednarik Roman. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empirical Software Engineering* 25 (2020), 3128–3174.

[29] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo Juzgado. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 666–676.

[30] Dario Salvucci and Joseph Goldberg. 2000. Identifying Fixations and Saccades in Eye-tracking Protocols. In *Proceedings of the Symposium on Eye Tracking Research & Applications (ETRA'00)*. 71–78.

[31] Reydne Bruno dos Santos. 2021. *Um Estudo sobre Definição e Avaliação da Readability e Legibility do Código Fonte*. Master's thesis. Universidade Federal de Pernambuco.

[32] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *Proceedings of the International Conference on Program Comprehension (ICPC'18)*. 31–40.

[33] Zohreh Sharafi, Yu Huang, Kevin Leach, and Westley Weimer. 2021. Toward an Objective Measure of Developers' Cognitive Activities. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 1–40.

[34] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. Women and Men—Different but Equal: On the Impact of Identifier Style on Source Code Reading. In *Proceedings of the International Conference on Program Comprehension (ICPC'12)*. IEEE, 27–36.

[35] Bonita Sharif and Jonathan Maletic. 2010. An Eye Tracking Study on Camelcase and Under_score Identifier Styles. In *Proceedings of the International Conference on Program Comprehension (ICPC'10)*. IEEE, 196–205.

[36] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education (TOCE'13)* 13, 4 (2013), 1–40.

[37] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. 2001. PEP8–StyleGuide for Python Code. *Python.org* 1565 (2001), 28.