



# Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension

José Aldo Silva da Costa<sup>1</sup> · Rohit Gheyi<sup>1</sup> · Fernando Castor<sup>2</sup> · Pablo Roberto Fernandes de Oliveira<sup>1</sup> · Márcio Ribeiro<sup>3</sup> · Baldoino Fonseca<sup>3</sup>

Accepted: 1 March 2023 / Published online: 18 May 2023  
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

## Abstract

Code comprehension is crucial for software maintenance and evolution, but it can be hindered by tiny code snippets that can confuse the developers, called atoms of confusion. Previous studies investigated how atoms impact code comprehension through the perspectives of time, accuracy, and opinions of developers. However, we need more studies evaluating other perspectives and the combination of these perspectives on a common ground through experiments. In our study, we evaluate how the eye tracking method can be used to gain new insights when we compare programs obfuscated by the atoms with functionally equivalent clarified versions. We conduct a controlled experiment with 32 novices in Python and measure their time, number of attempts, and visual effort with eye tracking through fixation duration, fixations count, and regressions count. We also conduct interviews and investigate the subjects' difficulties with the programs. In our results, the clarified version of the code with *Operator Precedence* reduced the time spent in the region that contains the atom to the extent of 38.6%, and the number of answer attempts by 28%. Most subjects found the obfuscated version more difficult to solve than the clarified one, and they reported the order of precedence to be difficult to validate. By analyzing their visual effort, in the obfuscated version, we observed an increase of 47.3% in the horizontal regressions count in the atom region, making its reading more difficult. The additional atoms evaluated revealed other interesting nuances. Based on our findings, we encourage researchers to consider eye tracking combined with other perspectives to evaluate atoms of confusion and educators to favor patterns that do not impact the understanding and visual effort of undergraduates.

**Keywords** Atoms of confusion · Code comprehension · Eye tracking

## 1 Introduction

When writing code, developers communicate their intent to other developers. Correctly interpreting their intent is crucial for the software maintenance and evolution processes. This

---

Communicated by: Janet Siegmund

✉ José Aldo Silva da Costa  
josealdo@copin.ufcg.edu.br; peregrino001@gmail.com

Extended author information available on the last page of the article

interpretation occurs through a process of code comprehension, in which a developer reads the source code, often written by another developer, and understands its behavior. However, the developer's interpretation of a piece of code can often differ from that of the one who wrote the code due to tiny patterns that can cause misunderstandings. These tiny patterns that can obfuscate the code and confuse developers causing them to misjudge the behavior of the code are called atoms of confusion (Gopstein et al. 2017; Gopstein et al 2018; Langhout and Aniche 2021) when it has been experimentally shown there are functionally equivalent alternatives that lead to better performance.

Previous studies have investigated the impact of atoms of confusion on code comprehension using multiple criteria such as time, answer correctness, and opinions. They compare code containing atoms likely to cause confusion, **obfuscated** code, to functionally equivalent code hypothesized to be less confusing, **clarified** code (Gopstein et al. 2017). For instance, Gopstein et al. (2017) presented a catalog of atoms of confusion for the C language and experimentally showed that obfuscated code increased confusion by reducing the accuracy of the subjects, in comparison to their accuracy when analyzing clarified code. Besides showing that atoms are often found in the C real projects, Medeiros et al. (2019) investigated the developers' opinions regarding the atoms of confusion. The subjects perceived obfuscated code to be more confusing. Combining accuracy and opinions, Langhout and Aniche (2021) found that developers made more mistakes and perceived obfuscated code to be more confusing and less readable in the Java language.

Notwithstanding, the combination of different perspectives can give additional insights on the impact of atoms of confusion on code comprehension. The subjects' answer correctness in an experiment can tell us the outcome of developers' performance, but not how or why they behaved in a certain way (Gopstein et al 2020). Combining number of attempts with time, we can have a better idea about the subjects' behavior. To investigate where in the code the subject spent most of her time, we can combine time, number of attempts, and opinions of the developers. This can help us infer where or why the subjects were confused. Gopstein et al. (2020) used the think-aloud method in which the subjects verbalize their thoughts while reading the code. Yeh et al. (2021) employed an Electroencephalograph (EEG) device to analyze the subjects' brain activities and evaluate an alternative way to understand code comprehension. Oliveira et al. (2020) used a remote eye tracker to study the impact of atoms on the subjects' focus of attention. However, they did not explore the potential of fixation duration, fixations count and regressions count. In the code, these three metrics measure how long one fixates over an element, how many times one fixates on the element, and how many times one goes back to previous lines of code, respectively. Combined with time and number of attempts, we can assess where, when, and infer why and how the atoms impact the subjects' visual effort. We need more studies evaluating distinct perspectives on a common ground through experiments.

In our study, we evaluate how the eye tracking method can be used to gain new insights in the context of atoms of confusion. We report an eye tracking study that evaluates how six atoms of confusion affect the code comprehension of 32 novices in Python. We consider as novices in Python undergraduate students who know how to program but have little experience with Python. We compare programs containing six atoms of confusion with functionally equivalent clarified versions aiming to observe how and to what extent they influence the performance of the subjects regarding time, number of attempts, and visual effort. We measure visual effort with three eye tracking metrics, namely, fixation duration, fixations count, and regressions count. Time and answer correctness have been employed before to measure effects on code comprehension (Malaquias et al 2017; Schulze et al 2013). In addition, the visual effort has been measured before with fixation duration, fixations count, and

regressions count (Busjahn et al 2011; Binkley et al 2013; Sharafi et al 2015). We analyze these metrics in the whole code as well as in the main Area of Interest (AOI). The AOI defines the region of the code that contains the atom or its corresponding clarifying version. In our study, we selected six atoms that occur in real projects: *Multiple Variable Assignment*, *True or False Evaluation*, *Conditional Expression*, *Operator Precedence*, *Implicit Predicate*, and *Augmented Operator*. Each program containing one of these obfuscating atoms has a functionally equivalent clarified version to be compared.

The addition of visual effort to the time and number of attempts measures contributed with another dimension to better understand novices' code comprehension and productivity. With eye tracking, we could investigate 1) how much time the subjects spend in a specific region, i.e., line or lines of the code that contain the atom and its clarifying version, 2) to what extent the presence of the atoms impacts the fixation duration, fixations count, and regressions count, and 3) how the presence of the atom impacts the code reading. These results have implications for the research community. With the visual effort dimension, we could see more nuances not observed in previous works. For instance, the region of the code that contains the atom *Multiple Variable Assignment* required 38.6% less time than its clarified version. Moreover, even perceiving the obfuscated and clarified versions as similar in terms of difficulty, splitting the assignments between two lines to make the code clearer led the subjects to make 60% more regressions inside the area of interest. By analyzing the visual regressions horizontally and vertically, we observed that the subjects tend to read the obfuscated code containing multiple variable assignments within the same line in a more direct manner. Furthermore, there were more eye movement transitions between the two split lines containing the assignments and the lines of code that later use them. Vertical regressions are not possible on a single line of code; however, with eye tracking, we could measure the extent of the impact of breaking the line on the vertical regressions and infer the visual effort on doing so.

These nuances encourage researchers to consider eye tracking to evaluate atoms of confusion with fine granularity. These results raise the awareness of educators. Introductory courses should be more carefully designed not to use obfuscated code by *Multiple Variable Assignment*, *True or False Evaluation*, or *Operator Precedence*. The code containing the atoms showed a negatively significant impact on the subjects' abilities to understand and required more visual effort. The implication of these results for practitioners and language designers' community for Python is that they should be more careful when using constructions to simplify the language that could possibly impair the novices' abilities to understand the code.

The main contributions of this study are:

- We present a controlled experiment using eye tracking with 32 novices in Python programming language to evaluate six atoms of confusion (Section 4);
- We adapted a set of four atoms to Python language in addition of two atoms candidates to the set (Section 4);
- We present and discuss the eye tracking method as a valuable method for evaluating atoms of confusion (Section 5).

This article is organized as follows: Section 2 presents the motivating example. Section 3 presents the definition of the study and Section 4 presents the methodology. Section 5 presents and discusses the results obtained and the interview with the subjects. Section 6 presents the threats to validity, Section 7 presents the related work, and finally, Section 8 concludes the study.

## 2 Motivating Example

Atoms of confusion are prevalent in open-source projects in C language. Investigating the presence of atoms of confusion in 50 C open-source software projects, Medeiros et al. (2019) found more than 109,000 occurrences of 11 out of 12 atoms considered in their study. Some of these projects, such as *Apache*, *OpenSSL*, and *Python* compiler, comprise more than 200,000 lines.

Atoms of confusion do occur in other languages as well, such as in Python, one of the most used languages for programming nowadays.<sup>1</sup> For instance, in Fig. 1(a), we illustrate a *Conditional Expression* found in the *SwiftShader* project for Python language and adapted to a complete code snippet. Iterating over the list of elements, in Line 4, `num` receives the value of `elem` if `elem` is equal to three. Otherwise, `num` receives one. If the implications of the study of Gopstein et al. (2017) for C language are sustainable for Python as well, the *Conditional Expression* in Line 4 impairs the code understanding of undergraduate students because the assignment depends on the value of a variable, which can confuse the student about the behavior of the code. Thus, the code in Fig. 1(a) can be obfuscated by the presence of the atom.

To clarify this source of confusion, Medeiros et al. (2019) proposed an alternative solution which we adapted for Python and presented in Fig. 1(b). In it, the line that contains the atom becomes four lines of code, and the variable `num` is used twice, depending on the condition of the `if` statement in line 4. Besides proposing a clarified version of the code that contains the atom of confusion, Medeiros et al. (2019) also investigated the subjective perception of experienced developers regarding the atom for the C language. Based on the answers of the developers, the code with the atom did not influence the understanding of the subjects negatively. In addition, the developers accepted pull requests with both obfuscated and clarified versions.

Langhout and Aniche (2021) derived a set of atoms of confusion based on the work of Gopstein et al. (2017), however, for Java language and performed a two-phase experiment with students investigating accuracy and perception. Out of 14 atoms, four presented results that were distinct from those presented in the study of Gopstein et al. (2017). One of these atoms that presented distinct results was the *Conditional Operator*.

In a more recent qualitative study, Gopstein et al. (2020) raised a serious concern. They pointed out that studies based on accuracy may be under-reporting the amount of misunderstanding. Just because the accuracy of answers is affected negatively, it is not always due to the factors the experiment was designed to test. Thus, we need to investigate this topic from a finer-grained perspective.

Oliveira et al. (2020) performed a controlled experiment including students and professionals that compared the obfuscated and clarified version of the code with atom *Conditional Operator*. However, Oliveira et al. (2020) investigated the objective performance of the developers solving tasks with both versions using an eye tracker. Their study did not find differences between the two versions regarding time, answer correctness, and visual attention in the AOI, agreeing with the developers' perception according to Medeiros et al. (2019). However, Oliveira et al. (2020) did not consider the fixation duration, fixations count, and regressions count to measure the extent of the impact of the atom on the visual effort of the subjects. Neither investigated how the atom affected the way the subjects read the code by distinguishing between horizontal and vertical transitions for all the subjects.

<sup>1</sup> [https://madnight.github.io/github/#/pull\\_requests/2021/4](https://madnight.github.io/github/#/pull_requests/2021/4)

```
1 elements = [7, 4, 3]
2 num = 0
3 for elem in elements:
4   num = elem if elem == 3 else 1
5 print(num)
```

(a)

```
1 elements = [7, 4, 3]
2 num = 0
3 for elem in elements:
4   if (elem == 3):
5     num = elem
6   else:
7     num = 1
8 print(num)
```

(b)

Fig. 1 Code adapted from *SwiftShader* with (a) obfuscated code containing the atom *Conditional Expression*, and (b) the clarified version of the code

To contribute to this debate, we need more empirical evidence from a finer-grained perspective to better understand which atoms of confusion can affect code comprehension and to what extent they do so. Since atoms are fine-grained code elements, coarser-grained approaches to assess code comprehension, e.g., correctness in tracing tasks, may be insufficient to capture their impact. A prior study showed the potential of eye tracking to investigate the effect of small-grained code changes on code comprehension (da Costa et al 2021). An eye tracker makes it possible to record the eye movements of human subjects and assess their visual attention (Rayner 1998). The eye tracking data allowed researchers to mainly assess visual attention and effort by investigating where the subject is fixating, the duration of their fixations, and how the fixations switched from one location to another (Sharafi et al 2015, 2020; Busjahn et al 2011).

For instance, we simplified a sequence of fixations performed by two subjects in Fig. 2. Each red circle represents a fixation that varies in size according to duration. The sequence and direction of fixations are depicted in chronological order with a number inside. In the obfuscated version (Fig. 2(a)), the subject makes eight fixations with six within the line of the atom (Line 4). In the clarified version (Fig. 2(b)), the subject makes five fixations with four of them within the atom region (Lines 4–7). Thus, the subject fixates more times and for a longer time in the obfuscated version. To perform eight fixations, the subject performs seven

```
1 elements = [7, 4, 3]
2 num = 0
3 for elem in elements:
4   num = elem if elem == 3 else 1
5 print(num)
```

(a)

```
1 elements = [7, 4, 3]
2 num = 0
3 for elem in elements:
4   if (elem == 3):
5     num = elem
6   else:
7     num = 1
8 print(num)
```

(b)

Fig. 2 Code with eye gaze patterns for (a) obfuscated code containing the atom *Conditional Expression*, and (b) the clarified version of the code

transitions. A transition consists of any eye movement between two fixations in any direction on the code. In the obfuscated code, we observe more visual transitions. In addition, the subject regresses visually in the code more times in the obfuscated version. While transitions are eye movements in any direction, regressions are a subset of transitions with a direction opposed to the code writing system. In the obfuscated version, she goes back three times in code, twice vertically examining the list, and one time horizontally to possibly inspect a variable. In the clarified version, the subject goes back only once to the list, making a vertical regression between lines. By examining their behavior at this small-grained level, we can see nuances not observed in previous works. Thus, besides measuring time and number of attempts, we investigate the effects of atoms of confusion on visual effort through fixation duration, fixations count, and regressions count.

### 3 Study Definition

In this section, we present the definition of our study following the Goal-Question-Metrics approach (Basili et al 1994). **We compare** programs containing obfuscating atoms with functionally equivalent clarified versions of these programs **for the purpose of** evaluating how the eye tracking method can be used to gain new insights **with respect to** 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 five research questions (RQs). Our null hypothesis for each RQ is that there is no difference between the obfuscated and clarified programs concerning the collected metric.

- **RQ: To what extent do the atoms affect task completion time?** To answer this question, following prior studies (Gopstein et al. 2017; de Oliveira et al 2020), we measure how much time it takes for the subject to specify the correct output of the task. In addition, we measure how much time the subject spends in specific areas of the code.
- **RQ: To what extent do the atoms affect the number of attempts?** To answer this question, also following prior studies (Gopstein et al. 2017; de Oliveira et al 2020), we measure the number of attempts by counting the number of attempts made by the subject until answering the task correctly.
- **RQ: To what extent do the atoms affect fixation duration?** To answer this question, we measure the duration of each fixation found in the captured data of the novices. In the code comprehension scenario, fixations with high duration have been associated with an increased level of attention (Busjahn et al 2011).
- **RQ: To what extent do the atoms affect fixations count?** To answer this question, we count all fixations found in the captured data of the novices. A high number of fixations has been associated with a longer time to process and understand code phrases (Binkley et al 2013), more attention to complex code (Crosby et al 2002), and more visual effort to recall identifiers' names (Sharafi et al 2012).
- **RQ: To what extent do the atoms affect regressions count?** Regressions in the context of natural language reading may indicate that the reader did not understand what they read (Rayner 1998). In the programming context, regressions have been used to assess the linearity of code reading (Busjahn et al 2015). In an imperative programming language, text lines may be read left-to-right, top-to-bottom, similarly to natural language. However, there are constructs, such as loops, that require the reader to read bottom-to-top at some points. To make the comparison fair, both obfuscated and clarified versions of the code have loops that iterate over the same number of elements. Thus, to measure regressions,

we compute the number of saccades with a direction opposed to the writing system, which can happen from a line of code to a previous one or within the same line. We compute the number summing all regressions across all attempts. To compare the programs, we compute the median number of regressions on each program.

## 4 Methodology

In this section, we present the methodology of our study. We present the pilot study (Section 4.1), experiment phases (Section 4.2), subjects (Section 4.3), treatments (Section 4.4), evaluated atoms of confusion (Section 4.5), programs (Section 4.6), eye tracking system (Section 4.7), fixation and saccades instrumentation (Section 4.8), and finally the analysis (Section 4.9).

### 4.1 Pilot Study

Before conducting the actual experiment, we conducted pilot studies with five human subjects. The purpose was to refine the material, such as forms and programs, and evaluate the experiment setup and design. We do not consider these five subjects in the analysis of the results.

Our study material comprises a set of programs, a form for characterizing the subjects, and questions for a semi-structured interview. To evaluate our set of programs, we tested complete code snippets from introductory programming courses. We validated the level of difficulty of the programs, code font size, font style, spaces between the lines of code, and indentation. In addition, we estimated the average time of each task, which allowed us to set a proper time limit for them. We found that each of our programs usually took less than two minutes to be solved. We also refined the questions from the forms.

Since our subjects are native Brazilians, we designed our programs and the vocabulary to be in the Brazilian Portuguese language, thus, avoiding problems in code comprehension given the lack of knowledge of words in English, for instance. We evaluated a limited vocabulary of words to name the variables in the programs. The identifiers were carefully selected, discussed by the researchers, and designed to convey some but not all of the information. For instance, we used words such as `elements` and `items`, which are general terms for lists of elements; `value`, `result`, and `total` for receiving the result of operations and printing the output; in addition, we used two abbreviated words such as `elem` and `cont` to specify an element and a counter, which are commonly employed in the context of teaching introductory programming languages. In specific programs, we also used words such as `grade`, `bonus`, `average`, and `final` to convey meaning in the specific context.

While previous empirical studies used variables with meaningless single-letter names (Gopstein et al. 2017, 2020; Langhout and Aniche 2021), we opted for names that conveyed some but not all the information. Meaningless names can make the code intrinsically harder to understand, and, therefore, differences are likely to be accentuated. However, in real code reading tasks, developers are usually, though not always, faced with variable names that use real words and have a meaning for them (Lawrie et al 2007). Therefore, following this approach is arguably closer to a practical scenario.

Through the conduction of the pilot studies, we learned that, when studying in the context of novices, we should provide the tasks in their mother tongue, otherwise comprehension would be hampered by natural language barriers. In addition, we should evaluate tasks with different levels of difficulty, which allows us to have a better set of tasks. Finally, we should

ask the subjects questions about the programs, identifiers' names, and other suggestions to refine the tasks.

The pilot studies allowed us to evaluate and refine our experiment design, which consists of four phases: (1) Tutorial, (2) Warm-up, (3) Task, and (4) Qualitative Interview. We then estimated an average of around 60 minutes for each subject to complete all phases. Next, we describe these phases in detail.

## 4.2 Experiment Phases

As the subject enters the room, we introduce ourselves and explain the main idea of the study, what data we are going to capture, and for what reasons. We asked each subject to fill out a consent form, agreeing to participate in the study and permitting the researchers to use the data for academic purposes only. Access to the collected data was restricted to the researchers, and the identity of the subjects was kept in anonymity.

In phase one, we present a tutorial with explanations regarding the execution of the experiment. All subjects reported being familiar with Python language, thus, we present some snippets to ensure they were familiar with them. We instruct the subjects on how to sit properly on the chair in front of the camera and how to perform the task. In addition, we explain that the subject has the option of quitting at any time and does not need to provide any reasons for doing that. Once the subject is seated comfortably in front of the camera, we explain how the camera calibration process works, and we proceed with a calibration of the camera on each subject's eye. In the camera calibration process, the subject looks at specific locations on the screen indicated by the camera software, and the same software indicates when calibration is successfully done. For some subjects, we had to re-calibrate the camera until we gained confidence that the data captured by the camera was reliable.

In phase two, we simulate the execution of the experiment with a simple warm-up task. While solving the task, we demonstrate how the subjects can specify the output, how the subject can close their eyes for two seconds before and after solving the task, how we signal the correct and incorrect answer, and how we signal the time limit. The idea is that the subject can be comfortable with the experiment setup and equipment.

In phase three, we run the actual experiment with twelve programs, six of them containing a distinct obfuscating atom each, and six functionally equivalent clarified programs. To avoid learning effects, we use a Latin Square design (Box et al 2005) for the experiment. We explain this in more detail in Section 4.4.

In phase four, we end the experiment with a semi-structured interview. The goal is to obtain qualitative feedback on how the subjects examined the programs and their subjective impressions. We go through each of the twelve programs and ask three questions: (1) How difficult was it to find the output: very easy, easy, neutral, difficult, or very difficult? (2) How did you find the output? (3) What were the difficulties you had, if any?

The coronavirus pandemic made the running of eye tracking experiments more challenging. It is worth mentioning that the health and safety of our subjects are of utmost importance to us. We started running the experiment only after the end of the country's social distancing measures, the infections were decreasing, and the number of vaccinated people was increasing. All subjects had at least one dose of a COVID-19 vaccine. Still, we arranged an environment with fresh air, and all subjects had Personal Protective Equipment (PPE) and disinfecting supplies such as hand sanitizers and face masks. Since we had one subject at a time, we limited the number of people in the environment to only two.



In addition to taking care of the environmental condition for the subjects' health, we were also careful with environmental aspects to reduce noise in the data. For instance, we did not use a swivel chair because, in previous pilot studies, subjects tended to move, which reduced the precision of the eye tracking equipment. Despite the measures we have taken, obtaining perfect data is virtually impossible, given camera limitations. Thus, we as researchers have discussed the collected data, plotted, interpreted, and performed data correction by slightly shifting chunks of fixations in the y-axis. We discuss in more detail this strategy in the threats to validity section (see Section 6.1). The dataset generated during the current study is available on Zenodo repository within a replication package also containing other materials (da Costa et al 2023).

### 4.3 Subjects

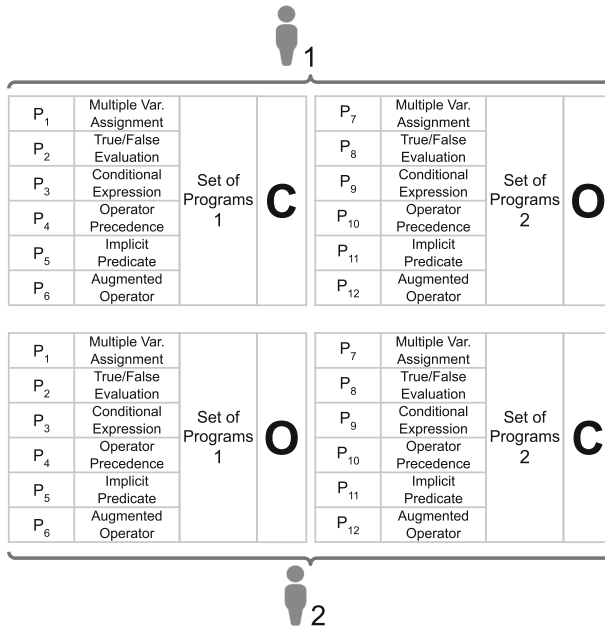
Our study included 32 undergraduate students that we call "novices". On average, our subjects had 20 months of experience with programming languages, including Python, Java, JavaScript, C, and C++. However, only in Python, the language in which the programs were written, they had on average seven months of experience. Thus, we refer to our subjects as novices in Python language. They were recruited from three universities in Brazil and were invited mainly through e-mails or text messages. All subjects were Brazilian Portuguese speakers enrolled in academic universities.

Regarding the sample size, we computed the number of subjects necessary to have a minimal power of 0.8 with a significant level of 0.05 using the T-test sample size computation. Our analysis revealed that we need 26 subjects in two samples to have a minimal power of 0.8 with a significant level of 0.05. Alternatively, since we have 32 subjects instead of 26, our study can also detect a moderate effect size of 0.5 with a power of 0.5 and a significant level of 0.05.

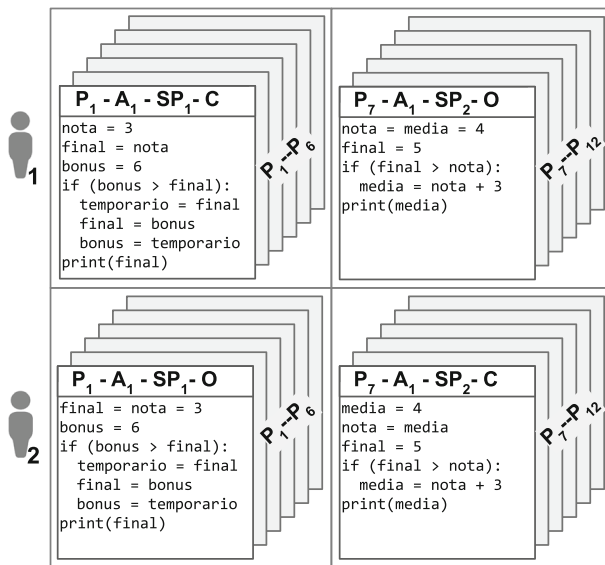
### 4.4 Treatments

Each subject has the task of examining 12 programs ( $P_1$ – $P_{12}$ ). To avoid the same subject taking a program in obfuscated and clarified versions, causing learning effects, we have designed 24 distinct programs divided into two sets of programs ( $SP_1$  and  $SP_2$ ), each containing 12 programs. A subject takes six obfuscated programs (O) of the first set, for instance,  $SP_1$ , and six clarified programs (C) of the second set,  $SP_2$ , as seen in Fig. 3. Both sets  $SP_1$  and  $SP_2$  have the same atoms; however, they are instantiated in distinct programs with distinct outputs. The obfuscated programs are our baseline group, and the clarified ones are the treatment group.

For instance, consider Fig. 4, which depicts how the atom *Multiple Variable Assignment* ( $A_1$ ) is evaluated in the programs  $P_1$  and  $P_7$  of the set of programs  $SP_1$  and  $SP_2$ , respectively. The first subject examines the program  $P_1$ , which is the clarified version of a program containing  $A_1$ , present in the set of programs  $SP_1$  ( $P_1 - A_1 - SP_1 - C$ ). The same subject examines another program,  $P_7$ , which contains the obfuscating atom  $A_1$ , present in the set  $SP_2$  ( $P_7 - A_1 - SP_2 - O$ ), which prints a distinct output from that one in  $SP_1$  to reduce learning effects. A similar idea applies to the second subject. The second subject examines  $P_1$  containing the obfuscating atom  $A_1$  present in  $SP_1$  ( $P_1 - A_1 - SP_1 - O$ ), and then the same subject examines another program,  $P_7$ , which is the clarified version of a program containing  $A_1$ , present in  $SP_2$  ( $P_7 - A_1 - SP_2 - C$ ). It is important to mention that being in the same set, the programs  $P_1 - A_1 - SP_1 - O$  prints the same output of  $P_1 - A_1 - SP_1 - C$ ; however, the first program contains the obfuscating atom and the second is a clarified version of the obfuscated program. Both



**Fig. 3** Structure of a Latin Square. Subject<sub>1</sub> takes six programs (P<sub>1</sub>–P<sub>6</sub>), which are clarified versions (C) of the programs containing each of the atoms. These programs are from Set of Programs 1 (SP<sub>1</sub>). Subject<sub>1</sub> also takes six programs (P<sub>7</sub>–P<sub>12</sub>) from the Set of Programs 2 (SP<sub>2</sub>) comprising the obfuscated code (O) containing the atoms. Subject<sub>2</sub> takes the complement to that



**Fig. 4** The structure of the programs P<sub>1</sub> and P<sub>7</sub>, whether obfuscated or clarified version, from SP<sub>1</sub> and SP<sub>2</sub>. We present all the programs with the obfuscating atoms and the clarified versions of the programs from SP<sub>1</sub> and SP<sub>2</sub> in our website (da Costa et al 2023)

**Table 1** Atoms evaluated in this study

Atoms	Description	Obfuscated	Clarified
Multiple Variable Assignment	We can assign the same value to multiple variables by using = consecutively	<code>a = b = 1</code>	<code>b = 1 a = b</code>
True or False Evaluation	Directly checking whether something is <code>True</code> versus checking whether its negation is <code>False</code>	<code>(not a == b)</code>	<code>(a != b)</code>
Conditional Expression	Sometimes called “ternary operator”, an <code>if</code> statement can be written in one line with conditional expression	<code>a = b if b == c else d</code>	<code>if (b == c): a = b else: a = d</code>
Operator Precedence	The precedence of the operators influences the outcome. Depending on the order, we might have different results	<code>a and b or c</code>	<code>(a and b) or c</code>
Implicit Predicate	An expression that does not produce a <code>bool</code> is used as a predicate	<code>(a % b)</code>	<code>(a % b != 0)</code>
Augmented Operator	A single operator adds a value to a variable and updates it	<code>a += 1</code>	<code>a = a + 1</code>

programs are examined by distinct subjects as well. In all the programs, the subjects had the task of specifying the correct output in an open-ended fashion, which means that each subject has to read the entire code and find the output without being presented with multiple options.

#### 4.5 Evaluated Atoms of Confusion

We evaluated six atoms of confusion, summarized in Table 1. We selected four of them, namely *Multiple Variable Assignment*, *Conditional Expression*, *Operator Precedence*, and *Implicit Predicate* from a popular catalog of atoms for C that was proposed by Gopstein (2017; 2018), and further adapted to Java (Langhout and Aniche 2021; Mendes et al 2021). We selected atoms that varied in their characteristics, which we found relevant for our investigation purpose, could be adapted for Python, and that were found in real projects. Gopstein et al. (2017) studied the prevalence of the atoms of confusion in 14 large and significant open-source projects. The four selected atoms in our study, based on their work, are among the seven most found atoms in the evaluated projects.

In their catalog, Gopstein et al. (2017) exemplify *Multiple Variable Assignment* as  $\forall 1 = \forall 2 = 3$ . This atom was originally named *Assignment as Value*. Nevertheless, in Python, assignments are not expressions, and this pattern has different semantics. It means that the same value is assigned to multiple variables. Due to the syntactic similarity and semantic difference, we use it as an atom and name it *Multiple Variable Assignment*.

In addition to those four atoms, we selected two other patterns that are found in style guides for Python language, namely *True or False Evaluation*<sup>2</sup> and *Augmented Operator*.<sup>3</sup> While planning the experiment, we came up with distinct atom candidates that had the potential to cause confusion. We gave preference to candidates commonly used in practice, comprising distinct characteristics, and we had alternative ways to write them. However, the pilot tests were crucial to determining what candidates would be more likely to cause confusion. We

<sup>2</sup> <https://google.github.io/styleguide/pyguide.html>

<sup>3</sup> <https://www.python.org/dev/peps/pep-0577/>

aimed to understand how our approach to evaluating those four atoms of confusion could be applied to evaluate these two atom candidates. In general, we conducted pilot tests to arrive at these six atoms through refinements, investigation of feasibility, and discussion among the authors.

## 4.6 Programs

We selected code snippets by manually analyzing code repositories of programming activities for introductory programming students. We mainly targeted GeeksForGeeks<sup>4</sup> and Leetcode,<sup>5</sup> which are popular code bases with programming activities for practicing and coding interviews. Given that we focused on novices, we selected easy, small, and complete problems and adapted them to the camera constraints. We present an example of our programs for each atom of confusion in Fig. 5.

A prior systematic literature review on code comprehension conducted by Oliveira et al. (2020) revealed that 70% of the studies in this domain involve asking subjects to provide information about the code, such as predicting the output. Following this commonly adopted methodology, we provided a code snippet to the subjects and asked them to specify the correct output. We are aware of other types of tasks in code maintenance and evolution, such as adding functionality, refactoring the code, and fixing bugs. However, we based this study on the assumption that the subject will need at least to know the code output or the state of the variables to perform these other activities.

We used programs with less than ten lines of code to fit completely on the screen. All the programs are free of syntactic errors. We used simple constructions that commonly occur in many programming languages. We ensured that each program contains exactly one or zero instances of one atom, whether the obfuscated or clarified version. Each program prints only one correct output given a set of possible outputs. We avoided letting the programs present only two possible outputs, even though, given the logic of the code, the code containing the *Conditional Expression* (Fig. 5(c)) and *Operator Precedence* (Fig. 5(d)) do that. We made sure that each version of the program, obfuscated and clarified one from the same set of programs, presented the same output. For instance, the right-hand side and left-hand side of Fig. 5(a). It is important to stress that, due to the use of a Latin Square design, no subject was presented with the clear and obfuscated versions of the same program. The programs followed Consolas font style, font size 16, line spacing of 1.5 inches, and eight white spaces of indentation.

## 4.7 Eye Tracking System

We used the Tobii Eye Tracker 4C in our experiment with a sample rate of 90 Hz. The calibration of the eye tracker followed the standard procedure of the device driver: while calibrating, the subject is asked to look at five points appearing one at a time randomly, twice, then, at the final stage, eight points appear for checking the calibration, three to the left, two in the middle, and three to the right. The eye tracker was mounted on a laptop screen with a resolution of 1366 x 720 pixels, a width of 30.9 cm, and a height of 17.4 cm at a distance of 50-60 cm from the subject. The code tasks were displayed as an image in the full-screen mode, but no Integrated Development Environment (IDE) was used, nor was a

<sup>4</sup> <https://www.geeksforgeeks.org/>

<sup>5</sup> <https://leetcode.com/>

Multiple Variable Assignment

<pre>final = nota = 3 bonus = 6 if (bonus &gt; final):     temporario = final     final = bonus     bonus = temporario print(final)</pre>	<pre>nota = 3 final = nota bonus = 6 if (bonus &gt; final):     temporario = final     final = bonus     bonus = temporario print(final)</pre>
---	--

(a) SP<sub>1</sub> - Obfuscated and clarified version

True or False Evaluation

<pre>valor = 0 cont = 1 while (cont &lt;= 4):     if (not cont == 3):         valor = valor + 1     cont = cont + 1 print(valor)</pre>	<pre>valor = 0 cont = 1 while (cont &lt;= 4):     if (cont != 3):         valor = valor + 1     cont = cont + 1 print(valor)</pre>
--	--

(b) SP<sub>2</sub> - Obfuscated and clarified version

Conditional Expression

```
elementos = [7, 4, 3]
resultado = 0
for elem in elementos:
    resultado = elem if elem == 3 else 10
print(resultado)
```

(c) SP<sub>1</sub> - Obfuscated and clarified version

```
elementos = [7, 4, 3]
resultado = 0
for elem in elementos:
    if (elem == 3):
        resultado = elem
    else:
        resultado = 10
print(resultado)
```

Operator Precedence

```
pontos = 15
if (False and True or True):
    media = pontos/3
else:
    media = 0
print(media)
```

(d) SP<sub>1</sub> - Obfuscated and clarified version

```
pontos = 15
if ((False and True) or True):
    media = pontos/3
else:
    media = 0
print(media)
```

Implicit Predicate

```
elementos = [7, 12, 10]
valor = 0
for elem in elementos:
    if (elem % 5):
        valor = valor + 1
print(valor)
```

(e) SP<sub>2</sub> - Obfuscated and clarified version

```
elementos = [7, 12, 10]
valor = 0
for elem in elementos:
    if (elem % 5 != 0):
        valor = valor + 1
print(valor)
```

Augmented Operator

```
elementos = [60, 30, 40]
limite = 50
total = 0
for elem in elementos:
    if (elem < limite):
        total += 1
print(total)
```

(f) SP<sub>1</sub> - Obfuscated and clarified version

```
elementos = [60, 30, 40]
limite = 50
total = 0
for elem in elementos:
    if (elem < limite):
        total = total + 1
print(total)
```

**Fig. 5** Examples of programs from the set of programs SP<sub>1</sub> and SP<sub>2</sub> with obfuscated (left-hand side) versions of the code containing the atoms *Multiple Variable Assignment*, *True or False Evaluation*, *Conditional Expression*, *Operator Precedence*, *Implicit Predicate*, and *Augmented Operator*, and their respective clarified (right-hand side) versions. Shaded areas represent the AOIs, which are the code lines in which both obfuscated and clarified versions differ

number for the lines. From this distance, we compute an accuracy error of 0.7 degrees which translates to 0.6 lines of inaccuracy on the screen, considering the font size we used and the line spacing. The line spacing was tested in the pilot study to be sufficiently large, so we could overcome the eye tracker accuracy limitations. For processing the gaze data, we implemented a script in Python, which allowed us to analyze and collect the metrics.

### 4.8 Fixation and Saccades Instrumentation

During a fixation, our visual attention is focused on a specific area of the stimulus and triggers cognitive processes (Just and Carpenter 1980). Thus, a fixation can be understood as the stabilization of the eye on part of a stimulus for a period of time, and the rapid eye movements between two fixations are called saccades (Salvucci and Goldberg 2000; Holmqvist et al 2011). The visual stimulus can be any object, for instance, a piece of source code, over which the subject performs a task and whose visual perception by the subject triggers cognitive processes and actions, such as edit of a statement in a source code file (Sharafi et al 2020).

There is no standardized threshold in the literature to specify the exact period of time for a fixation because duration usually depends on the processing demands of the task. However, we have some guidelines popular among eye tracking researchers. Salvucci and Goldberg (2000) define a fixation as the eye being stable for a period of time between 100 and 200 ms, while according to Rayner (1998), our eyes remain relatively still during fixations for about 200–300 ms when reading natural language text. Thus, after analyzing our programs, we used 200 ms as our threshold. Eye tracking researchers usually use an algorithm to classify gaze samples into fixations based on this threshold.

In this study, we used a Dispersion-Based algorithm to classify the fixations. In particular, we used the Dispersion-Threshold Identification (I-DT) (Salvucci and Goldberg 2000). We also classified gaze samples as belonging to a fixation if the samples are located within a spatial region of approximately 0.5 degrees (Nyström and Holmqvist 2010). This region corresponded to 25 pixels in our screen.

#### 4.9 Analysis of the Results

From the total of 384 programs, the subjects solved 329, corresponding to 85.6% of the programs. This set of solved programs includes programs that were solved either in the first attempt or after many attempts. However, both were solved within the time limit. We based our analysis only on these solved programs within the time limit. From this total, 160 programs were obfuscated, and 169 were clarified.

Due to technical issues with the camera, we missed the data for two programs. They consist of only 0.5% of the data. Since they were not associated with correct or incorrect answers, and as a requirement of the statistical analysis based on the Latin Square design, we decided to impute them. Aiming to impute missing data for two programs, we used the Multivariate Imputation by Chained Equations (MICE) method implemented as a mice package in R for multiple imputations, namely Predictive Mean Matching (PMM). The PMM method uses the predictive mean matching (Jadhav et al 2019) to impute univariate missing data. This method performs better when the data sample size is sufficiently large (Kleinke 2017), which was our case.

The eye tracker equipment has limitations, such as calibrating the eyes of the subjects. We carefully calibrate and even re-calibrate when necessary. However, we still saw a need for an adjustment in the gaze points. We adopted the following strategy. We selected programs with a long horizontal line of code, specifically with lines that the subjects mentioned they were looking at in the interviews. Then we systematically analyzed whether the fixations plotted and heatmaps were on white spaces close to that long line, which could suggest a need for an adjustment. For certain subjects, the heatmap revealed a red color over a blank area not touching the code. Similarly, the plot of the fixation points sometimes revealed points over blank areas. For these cases, a small adjustment was sufficient to get the data corrected. The error for these cases was systematic, meaning that all the fixations for a particular program received the same adjustment. We discuss involved threats in Section 6.1.

Once we had collected and adjusted our data, we performed a statistical analysis to test our null hypotheses. We determine a significance level of 0.05, which means a 5% risk of concluding that there is a difference when there is no actual difference. Whenever our  $p$ -value is equal or inferior to 0.05, we reject the null hypothesis that there was no difference between the median of the treatments.

We used statistical tests to compare two groups regarding the time, number of attempts, and visual metrics. The obfuscated programs are the control group, and the clarified programs are

our treatment group. Following a practical guide on eye tracking studies (Sharafi et al 2020), to test if data were normally distributed, we used Shapiro-Wilk Test (Shapiro and Wilk 1965). For the normally distributed data, we performed the parametric  $t$  test for the two independent samples. The  $t$  test can be used to verify whether there is a statistically significant difference between two groups (Sheskin 2020; Sharafi et al 2020). However, before performing the  $t$  test, we verify whether the variances of the two groups were equal (Sheskin 2020). For the data that do not follow a normal distribution and that could not be normalized, we used the non-parametric test Mann-Whitney, also known as Wilcoxon test, which can be applied to these specific situations (Sheskin 2020; Sharafi et al 2020). The mean value in the data might not be appropriate to characterize the fixations because the central tendency might depend on some very high values (Galley et al 2015). Thus, we based on the median as a measure of central tendency. To compare the six levels of atoms (six groups), we used ANOVA for normal data and Kruskal-Wallis for not normal data. We used the post-hoc Dunn's Test with p-values adjusted with the Bonferroni method to identify which groups were different.

To identify code reading patterns in our data, we adopted the following methodology: we identified a set of regions in the code, such as variable definition, loop condition, if condition, and others. Then, we defined these regions in pixels on the images of the tasks. We defined the regions inside the code according to a previous guideline (Holmqvist et al 2011). The regions were driven by our hypothesis; their positioning was precisely defined with 10 pixels to the right, left, top, and bottom, considering the camera limitations and so that we could have a margin between the regions, and the regions did not overlap. In addition, we defined the white-space as a region so that we could be aware of any threat to validity given the camera limitations. Using the chronological order of the fixations and their positions, we identified a sequence of visited regions for each participant. We then built a big picture of the sequences by simplifying repeated transitions from one region to the same region. We make the sequences and the images of the tasks with the regions identified available in our supplementary material (da Costa et al 2023).

## 5 Results and Discussion

In this section, we present our results for each atom and discuss them. We present the *Multiple Variable Assignment* (Section 5.1), *True or False Evaluation* (Section 5.2), *Conditional Expression* (Section 5.3), *Operator Precedence* (Section 5.4), *Implicit Predicate* (Section 5.5), and *Augmented Operator* (Section 5.6). We also present the coding of the subjects' answers (Section 5.7).

### 5.1 Multiple Variable Assignment

In Fig. 6, we depict the obfuscating atom *Multiple Variable Assignment* on the left-hand side and the clarified version on the right-hand side. They differ in that the clarified version

<b>Obfuscated</b>	<b>Clarified</b>
<code>final = nota = 3</code>	<code>nota = 3 final = nota</code>

**Fig. 6** Obfuscating atom *Multiple Variable Assignment* and clarified version

has two lines of code in the AOI, and one variable is repeated. In Table 2, we consider two perspectives of the metrics, one examining only the AOI and the other examining the whole code. While the time in the code, for instance, consists of the time one requires to examine and solve the task regardless of the fixations made, the time in AOI consists of the time examining only the region of the atom.

Concerning our RQ<sub>1</sub>, the subjects spent more time in the AOI of the clarified version. The clarified version has one more element to observe and one line to go back, which can explain the need for more time, more fixations, and more regressions examining. To better understand it, we investigated it by distinguishing between a regression to a previous line, vertical regression, and a regression within the same line, horizontal regression.

Concerning our RQ<sub>5</sub>, the clarified version presented slightly more vertical regressions in the code and fewer horizontal ones when considering the two sets of programs. In Fig. 7, we depict an example of the distribution of the regressions for two subjects who examine a program of SP<sub>1</sub>. We selected subjects whose patterns hold for all subjects. One takes the code with the atom *Multiple Variable Assignment*, and the other takes its respective clarified version of the code. In the graph, each edge represents a regression with a direction to a previous line of code or to the same line. Each node represents a line of code. The grayscale intensity of the edge represents the number of times such regression was repeated. Adding one more line might explain the increase in the vertical regressions. The reduction in the number of horizontal regressions in the AOI might be because, in the clarified version, we have two lines, but they are shorter compared to the obfuscated version. When we consider the sum of vertical and horizontal regressions, the clarified version presents more regressions.

To deepen our analysis and look for reading patterns in our programs, among our subjects, we identified a set of regions in the code as in Fig. 8. The colors distinguish between distinct regions and are identified by names such as `AssignedValue`, `IfCondition`, and `PrintOutput`. We analyzed the chronological order of the regions fixated by the subjects.

In the clarified version, the subjects go back and forth between the two lines to observe the same variable. Half of the subjects make the transition `Grade` → `GradeAssignment` 16 times. Half of the subjects also go back making the transition `GradeAssignment` → `Grade` 12 times. This may indicate confusion, given that the subjects have difficulty associating the same variable between different lines or need to remember the assigned value. Of the two subjects who failed to solve the task, on average, they went back and forth between `Grade` and `GradeAssignment` six times. Most of the subjects make the transition `FinalAssignment` → `GradeAssignment` in the clarified version more than necessary. The subjects may forget or make incorrect associations with the variable `FinalAssignment`, which is used five times in the code, and, in one of them, the variable is updated. When the task requires more use of temporary memory, the subjects need to go back in the code to refresh their memory. On average, 52% of subjects return from the lines that later use `FinalAssignment` to it.

We used a five-point scale to assess the subjects' opinions concerning how difficult they perceived the programs to be solved. We asked the subjects to rate each program individually, whether they found it very easy, easy, neutral, difficult, or very difficult. In Fig. 9, we compare their perceptions with obfuscated and clarified versions of the code containing the evaluated atoms.

The subjects perceive the obfuscated and clarified versions of the code as similar in terms of difficulty, according to Fig. 9(a). We observe a discrepancy between how subjects subjectively perceive the difficulty of the task and their performance on it. Such disagreement can be explained by the fact that self-evaluation of difficulties can be an intrinsically difficult activity. Not all subjects might be aware of their own effort while performing a task, such



**Table 2** Results for all metrics for all atoms. O = obfuscated code; C = clarified code; PD = percentage difference; PV = p-value; ES = effect size (Cliff's delta). Columns O and C are based on the median as a measure of central tendency, except for attempts, which is based on the mean

Atoms	Metrics	In the AOI				In the Code					
		O	C	PD %	PV	ES	O	C	PD %	PV	ES
Multiple Variable Assignment	Time (sec)	8.3	10.9	↑30.1	<b>0.03</b>	0.31	41.1	44.3	↑7.6	0.85	n/a
	Attempts	n/a	n/a	n/a	n/a	n/a	1.19	1.07	↓10.0	0.16	n/a
	Fix. Duration (sec)	4.2	5.1	↑22.9	0.15	n/a	19.4	18.2	↓6.1	0.93	n/a
	Fix. Count	13.0	18.0	↑38.4	0.17	n/a	59.5	59.5	0.0	0.86	n/a
	Reg. Count	2.5	4.0	↑60.0	<b>0.04</b>	0.29	26.0	25.0	↓3.8	0.67	n/a
True or False Evaluation	Horiz. Reg. Count	2.5	2.0	↓20.0	0.98	n/a	11.5	10.0	↓13.0	0.95	n/a
	Vert. Reg. Count	0.0	2.0	↑Inf	0.000	-	14.0	14.5	↑3.5	0.42	n/a
	Time (sec)	20.3	16.3	↓19.6	0.24	n/a	61.2	55.9	↓8.5	0.93	n/a
	Attempts	n/a	n/a	n/a	n/a	n/a	1.25	1.37	↑9.6	0.36	n/a
	Fix. Duration (sec)	10.1	10.6	↑4.7	0.47	n/a	35.4	27.0	↓23.7	0.81	n/a
Conditional Expression	Fix. Count	30.5	28.0	↓8.2	0.22	n/a	92.0	79.0	↓14.1	0.77	n/a
	Reg. Count	9.5	5.0	↓47.3	<b>0.03</b>	-0.34	41.5	35.0	↓15.6	0.74	n/a
	Horiz. Reg. Count	9.5	5.0	↓47.3	0.03	-	23.5	15.0	↓36.1	0.58	n/a
	Vert. Reg. Count	0.0	0.0	n/a	n/a	n/a	17.0	16.0	↓5.8	0.87	n/a
	Time (sec)	30.7	32.1	↑4.3	0.84	n/a	71.6	62.7	↓12.3	0.24	n/a
Conditional Expression	Attempts	n/a	n/a	n/a	n/a	n/a	1.22	1.14	↓8.8	0.46	n/a
	Fix. Duration (sec)	18.9	14.0	↓25.5	0.44	n/a	34.3	26.9	↓21.5	0.18	n/a
	Fix. Count	59.0	41.0	↓30.5	0.33	n/a	107.0	78.0	↓27.1	0.21	n/a
	Reg. Count	15.0	14.0	↓6.6	0.71	n/a	43.0	35.0	↓18.6	0.38	n/a
	Horiz. Reg. Count	14.0	8.0	↓42.8	0.01	-	26.0	14.0	↓46.8	0.02	-
Conditional Expression	Vert. Reg. Count	0.0	6.0	↑Inf	0.000	-	17.0	21.0	↑23.5	0.45	n/a

Table 2 continued

Atoms	Metrics	In the AOI				In the Code					
		O	C	PD %	PV	ES	O	C	PD %	PV	ES
Operator	Time (sec)	20.2	12.4	↓38.6	<b>0.009</b>	-0.37	43.5	34.7	↓20.1	<b>0.04</b>	0.29
	Attempts	n/a	n/a	n/a	n/a	n/a	1.62	1.16	↓28.3	<b>2x10<sup>-4</sup></b>	-0.46
	Fix. Duration (sec)	11.0	7.3	↓34.1	<b>0.02</b>	-0.33	19.9	17.1	↓14.1	0.08	n/a
	Fix. Count	32.5	22.0	↓32.3	<b>0.02</b>	-0.32	57.5	49.0	↓14.7	0.07	n/a
	Reg. Count	10.0	5.0	↓50.0	<b>0.02</b>	-0.33	25.0	19.0	↓24.0	0.06	n/a
Precedence	Horiz. Reg. Count	10.0	5.0	↓50.0	0.02	-	14.0	10.0	↓28.5	0.04	-
	Vert. Reg. Count	0.0	0.0	n/a	n/a	n/a	11.5	9.0	↓21.7	0.09	n/a
	Time (sec)	26.4	17.3	↓34.4	0.29	n/a	71.2	47.6	↓33.1	0.10	n/a
	Attempts	n/a	n/a	n/a	n/a	n/a	1.42	1.18	↓16.9	0.16	n/a
	Fix. Duration (sec)	16.1	11.0	↓31.5	0.42	n/a	35.1	25.9	↓26.2	0.33	n/a
Implicit Predicate	Fix. Count	43.0	29.0	↓32.5	0.45	n/a	86.0	64.5	↓25.0	0.28	n/a
	Reg. Count	10.0	8.0	↓20.0	0.88	n/a	40.0	28.5	↓28.7	0.26	n/a
	Horiz. Reg. Count	10.0	8.0	↓20.0	0.88	n/a	22.0	14.5	↓34.0	0.44	n/a
	Vert. Reg. Count	0.0	0.0	n/a	n/a	n/a	21.0	12.0	↓42.8	0.13	n/a
	Time (sec)	7.2	5.9	↓17.6	0.18	n/a	45.7	36.5	↓20.0	0.30	n/a
Augmented Operator	Attempts	n/a	n/a	n/a	n/a	n/a	1.13	1.20	↑6.1	0.69	n/a
	Fix. Duration (sec)	4.4	2.8	↓35.4	0.09	n/a	20.4	16.4	↓19.3	0.29	n/a
	Fix. Count	11.5	8.0	↓30.4	0.22	n/a	58.5	47.5	↓18.8	0.41	n/a
	Reg. Count	2.0	1.0	↓50.0	0.36	n/a	24.0	17.0	↓29.1	0.31	n/a
	Horiz. Reg. Count	2.0	1.0	↓50.0	0.36	n/a	9.0	6.0	↓33.3	0.08	n/a
Vert. Reg. Count	0.0	0.0	n/a	n/a	n/a	13.0	11.0	↓15.3	0.71	n/a	

p-values set in boldface indicate statistical significance (p-value < 0.05)

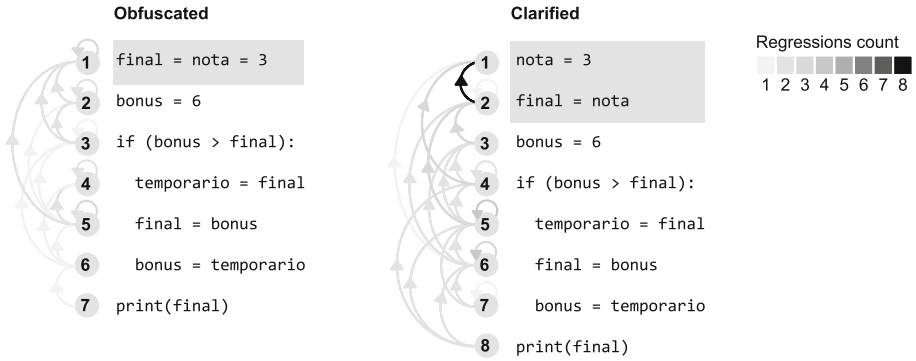


Fig. 7 Regressions graph with two subjects visually regressing horizontally and vertically while examining the code, one with the code containing the obfuscating atom *Multiple Variable Assignment* and the other with the clarified version of code

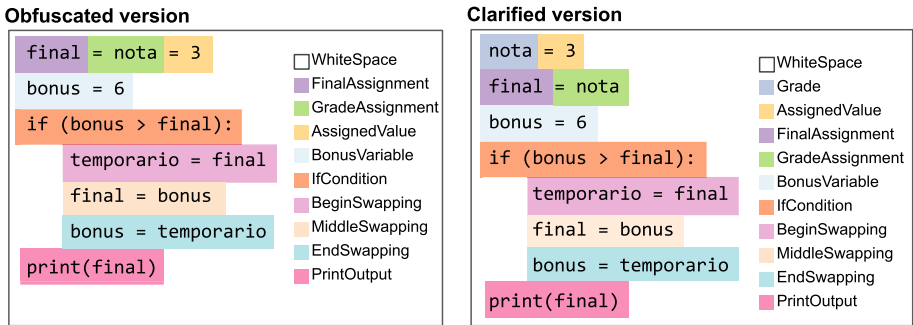


Fig. 8 Set of regions inside the code version with *Multiple Variable Assignment* atom and in the code with the clarified version of code

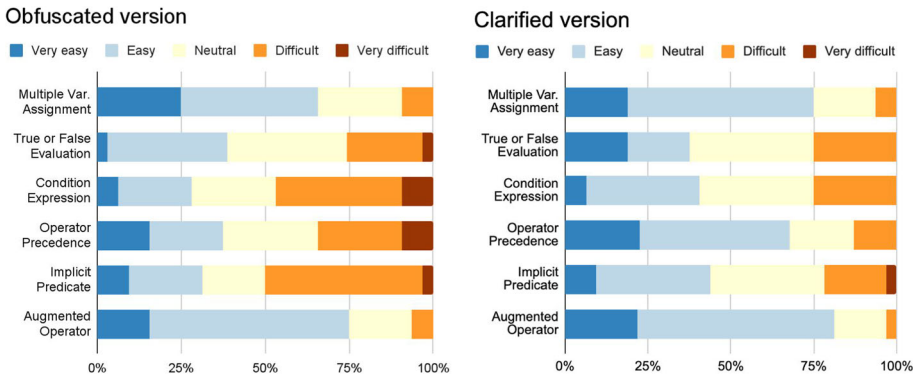


Fig. 9 Perception of difficulties with obfuscated code containing the evaluated atoms and their clarified version

as going back and forth in the code or remembering the variables' intermediate states. In this scenario, we triangulate the subjective feedback with the other perspectives to have a better comprehension of the phenomena of comprehension. Regarding the difference in time in the AOI and in the Code, we observed that, in the clarified version, the subjects mentioned difficulties with memorizing variables and difficulties with swapping values in the if condition. This might indicate that repeating the variable in the atom region may influence both their opinions and the variables' associations in the swapping region code outside AOI.

We conducted semi-structured interviews mainly to identify the subject's difficulties, better interpret the results, and perform minor sanity tests. In the obfuscated version, the subjects mentioned the issues: “*first line is confusing*”, “*first line caused me trouble*”, “*first line is hard*”, “*many variable assignments*”, and “*beginning strange*”. In the clarified version, the subjects mentioned: “*many variables*” and “*if is confusing*”. As a takeaway, in the obfuscated version, the sources of confusion concentrate on the first line with the atom, while in the clarified version, they concentrate on the number of variables.

Answering our RQs, in the clarified version of code with *Multiple Variable Assignment*, there is an increase in the time in the AOI (RQ<sub>1</sub>) and in the number of vertical regressions between the two lines (RQ<sub>5</sub>). In the obfuscated version, subjectively, the sources of confusion concentrate on the first line with the atom, while in the clarified version, they concentrate on the number of variables.

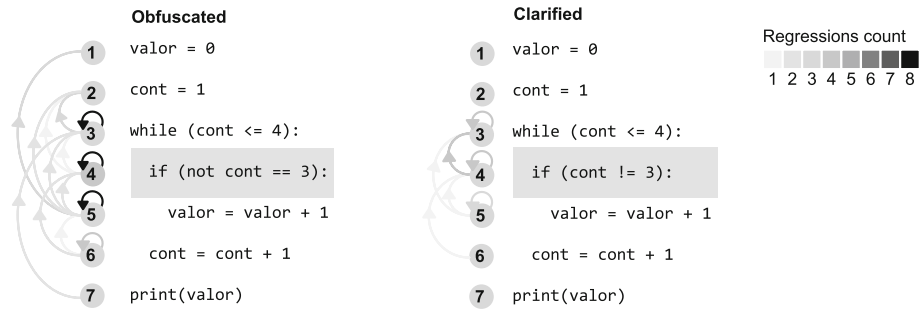
## 5.2 True or False Evaluation

In Fig. 10, we depict the obfuscating atom *True or False Evaluation* on the left-hand side and the clarified version on the right-hand side. The clarified version removes the `not` operator and replaces the equality operator (“`==`”) by a not equals (“`!=`”) operator.

Even though the time (RQ<sub>1</sub>) and the number of submissions (RQ<sub>2</sub>) presented a slight impact, the most impacted metrics were the visual metrics, especially the regressions count (RQ<sub>5</sub>). In terms of visual regressions, the clarified version reduced the median number of horizontal ones in the AOI as well as the number of entries and exits from the AOI. For instance, the obfuscated version of the program has 50% more horizontal regressions in the code than the clarified one. Within the AOI, the obfuscated version has almost twice as many horizontal regressions. In Fig. 11, we depict an example of the distribution of the regressions of two subjects on a program of SP<sub>2</sub> with the code containing the *True or False Evaluation* and the clarified version. Besides having more elements to observe horizontally, in the obfuscated version, the subjects have to check whether the right-hand side and the left-hand side are equal

<b>Obfuscated</b>	<b>Clarified</b>
<code>if (not cont == 3):</code>	<code>if (cont != 3):</code>

**Fig. 10** Obfuscating atom *True or False Evaluation* and clarified version

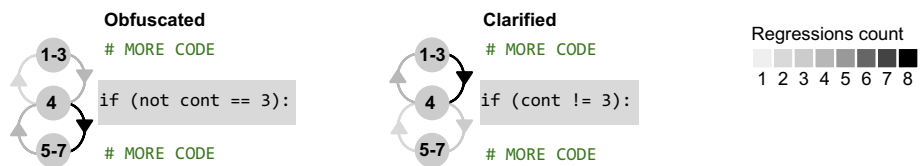


**Fig. 11** Regressions graph with two subjects visually regressing horizontally and vertically in the code, one with the obfuscated version and the other with the clarified version of the code containing the atom *True or False Evaluation*

and then apply the `not` operator. An interpretation for the same-line regressions for non-AOI lines in only the obfuscated version is that the variable in the AOI depends on the repetitive control structure of the loop before, which affects the incrementing variable outside the AOI after. If the AOI is confusing, one can make more same-line regressions in regions outside the AOI as well.

In terms of visual transitions, the subjects present a similar number between the AOI and the rest of the code for the obfuscated code containing the *True or False Evaluation* and the clarified version. We distinguish between transitions and regressions in the sense that transitions are eye movements with any direction in the code, while regressions are a subset of transitions with a direction opposed to the code writing system. While regressions are represented as the backward arrows in Fig. 11, a transition describes eye movements represented as forward and backward arrows such as in Fig. 12. We depict the transitions in the transition graphs because we aimed to examine how many times the subjects enter and exit the AOI. The median number of times the subjects visually enter the AOI is the same for both versions. Nevertheless, in the clarified version, the subjects transition more with the upper part of the code. In Fig. 12, we give an example of these transitions with two subjects who examine a program of  $SP_2$  with *True or False Evaluation*, one subject in each version. To convey the concept more concisely, the transitions between the lower part of the code and the upper part of the code are not present in the graph. In the obfuscated version, the subjects reported difficulties with the operator `not`, and the subject seems to visit more the lower part to make sense of it. In the clarified version, the subjects mentioned the increment and the loop, and they seem to visit the upper part with the `while` statement more times.

In Fig. 13, we depict the *True or False Evaluation* atom and its clarified version. We observed that for the obfuscated program, the addition of the particle `not` changes the visual dynamics and possibly the way of understanding. For instance, the region



**Fig. 12** Transitions graph with two subjects visually entering and exiting the AOI in the code, one with the obfuscated code containing the *True or False Evaluation* and the other with the clarified version of the code

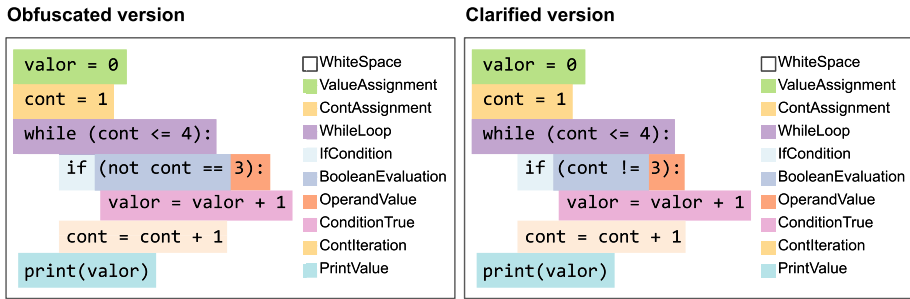


Fig. 13 Set of regions inside the code version with *True or False Evaluation* atom and in the code with the clarified version of code

BooleanEvaluation is crucial for the understanding of the programs in both versions, however, the subjects look at this region 45% fewer times in the clarified version compared to the obfuscated one. In addition, they regress to regions ContAssignment and WhileLoop about 50% fewer times. They also look at the region ConditionTrue 28% fewer times in the clarified version, which depends on the correct interpretation of the region BooleanEvaluation. Thus, we observe less effort from the subjects in moving between different regions to solve the tasks in the clarified version. While some subjects mentioned having difficulties understating the not particle in region BooleanEvaluation in the obfuscated version, none of them mentioned making wrong associations regarding the order of precedence.

For the obfuscated version of the atom *True or False Evaluation*, adding the particle not before the variable followed by equals might lead to confusion. The subjects might not be certain whether the not negates the variable or the expression of the variable followed by equals and a number. For instance, consider the transitions exhibited by the subjects in Fig. 14.

In the gaze transitions, we observed several transitions going forward and backward between not and the variable, which may be associated with confusion. Many subjects reported being confused about the not particle, including Subject 9, who needed two submissions to solve the task and presented time in the AOI twice the average. Confusion can be due to the lack of clear understanding about whether the variable or the equals operator gets negated. On the other hand, the clarified code with *True and False Evaluation* seemed to ease the operators' understanding. For instance, consider the transitions exhibited by the subjects in Fig. 15.

We observed fewer transitions going forward and backward in the expression. When we remove the not particle to clarify the code, the reading seems to follow the flow of the code better, decreasing the number of backward transitions one needs to make in the same line. Indeed, Subject 6 reported that "it was clear that the if statement was executed only once" and her time in the AOI was almost half of the time of the average of the subjects with the obfuscated version and the number of regressions about one-third of the average of the subjects with the obfuscated one.

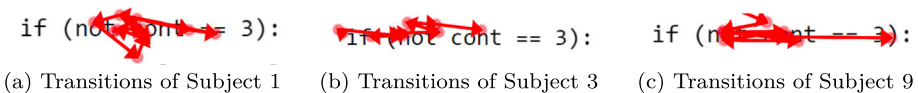


Fig. 14 Sequence of transitions of subjects on the obfuscated code version with *True or False Evaluation*

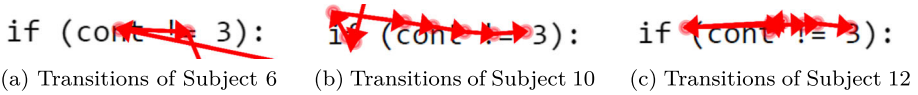


Fig. 15 Sequence of transitions of subjects on the clarified code version with *True or False Evaluation*

In the obfuscated version, the subjects mainly mentioned the following issues: “difficulties with not”, “not made it complicated”, “didn’t understand not”, “not is strange”, “if complex”, and “not is confusing”. For the clarified version, they mentioned: “increment confusing”, “difficulties with if”, and “confused the loop”. As a takeaway, in the obfuscated version, the sources of confusion are more concentrated in not, while in the clarified version, the sources are more diverse.

Answering our RQs, in the clarified version of code with *True or False Evaluation*, there is a reduction in the number of horizontal regressions in the AOI (RQ<sub>5</sub>). The additional metrics in RQ<sub>1</sub>–RQ<sub>4</sub> did not present a substantial impact. In the obfuscated version, subjectively, the sources of confusion are more concentrated in not, while in the clarified version, the sources are more diverse.

### 5.3 Conditional Expression

In Fig. 16, we depict the obfuscating atom *Conditional Expression* on the left-hand side, and the clarified version on the right-hand side. They differ in that the clarified version has three more lines of code in the AOI than the obfuscated version and more elements, such as the repetition of one variable. Concerning the time (RQ<sub>1</sub>) and number of submissions (RQ<sub>2</sub>), we observed a slight impact. However, with the clarified version, we observed substantial reductions in the duration of the fixations, in the fixations count, and in the horizontal regressions count (RQ<sub>3</sub>–RQ<sub>5</sub>).

We investigated all regressions from anywhere in the AOI to the for loop in both versions. We found that the subjects often regress from the AOI to the region ForLoop. Indeed, they regress 80% more times in the clarified version than in the obfuscated one. In the obfuscated version, most of the regressions go from region ConditionTrue to the ForLoop, while in the clarified version, most of them go from the region IfCondition to the ForLoop. We learned that most regressions to the ForLoop come from its subsequent regions in both versions. On the other hand, in the clarified version, the subjects make fewer regressions between the regions inside the AOI. In the clarified version, they return 31% fewer times between the regions compared to the obfuscated.

Concerning the number of submissions (RQ<sub>2</sub>) and the correctness of the answers, the obfuscated version was associated with more programs that were not solved. With the obfus-

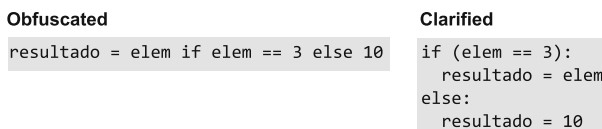
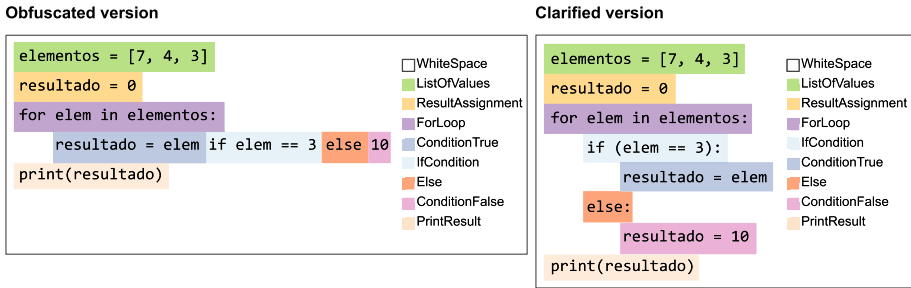


Fig. 16 Obfuscating atom *Conditional Expression* and clarified version



**Fig. 17** Set of regions inside the code version with *Conditional Expression* atom and in the code with the clarified version of code

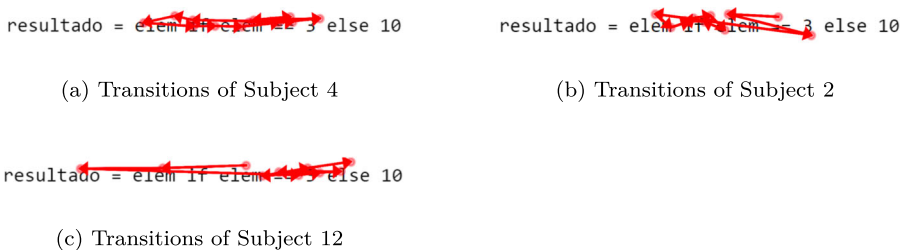
cated version, seven programs were not solved, while in the clarified one, only two were. For instance, concerning the program in Fig. 17, a subject who could not solve it exhibited the sequence *IfCondition* → *ConditionTrue* considerably more times than necessary. This behavior of returning several times may indicate that the type of structure adopted in the obfuscated version can confuse the subject, making him/her return unnecessarily repetitively. Similarly, the sequence *ForLoop* → *ConditionTrue* is made substantially more times than necessary.

One of the reasons why we have larger differences in the code than AOI can be that the clarified version makes the subjects go fewer times to the outside of the AOI. For instance, in *Conditional Expression*, in the clarified version, the subjects go 22% fewer times to the top where we have the variables declared.

The subjects perceive the obfuscated version as more difficult to understand. Quantitatively, we observed an alignment between perception or difficulty and actual results. The subjects needed more time and exhibit more visual effort with the obfuscated version.

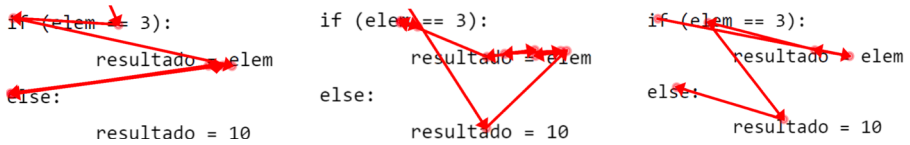
For the obfuscated version of the atom *Conditional Expression*, we observed several transitions going forward and backward in the center of the expression. Several subjects mentioned being confused about the condition line. For instance, Subject 4 mentioned that the ternary *if* was confusing and needed two submissions to solve the task while spending more than twice the average time in the AOI that the subjects needed in the clarified version. The fact that the true condition comes before the condition being tested can make the subjects go back more times to observe it. For instance, consider the transitions exhibited by the subjects in Fig. 18.

In the gaze transitions, the subjects often go back in the code to the true condition. Subject 4 regresses three times the average number of regressions in the AOI in the clarified



**Fig. 18** Sequence of transitions of subjects on the obfuscated code version with *Conditional Expression*





(a) Transitions of Subject 3 (b) Transitions of Subject 5 (c) Transitions of Subject 9

**Fig. 19** Sequence of transitions of subjects on the clarified code version with *Conditional Expression*

version, which can indicate confusion. We also observed similar transitions going forward and backward for other subjects. On the other hand, in the clarified code, we observed longer transitions between true and false conditions. For instance, consider the transitions exhibited by the subjects in Fig. 19.

The subjects make transitions between statements when the condition is true and false. However, unlike the obfuscated code, the transitions are not concentrated nor go back and forth so often. Instead, they are in the vertical between lines of code. Subject 5 commented about the structure of the code which she found easy. She solved the tasks in the first submission with half of the time and half of the number of regressions in the AOI compared to the average time and number of regressions in the obfuscated version. Breaking one long line of the condition expression into more lines modified the structure of the code but also seemed to improve the performance of the subjects.

The main issues mentioned by the subjects for the obfuscated version were: “*conditional expression is confusing*”, “*more time to validate if conditional*”, “*if conditional is complicated*”, “*unsure about if conditional*”, “*didn’t understand the list*”, “*didn’t remember if conditional*”, and “*if difficult and I prefer another style*”. For the clarified version, they mentioned: “*for loop*”, “*if condition is difficult*”, “*difficulties with elem*”, and “*indentation confusing*”, “*else difficult*”. As a takeaway, in the obfuscated version, confusion sources are more concentrated in the conditional expression. In the clarified version, the sources are more concentrated in variables and the condition of the if statement.

Answering our RQs, in the clarified version of code with *Conditional Expression*, there is an increase in the number of vertical regressions in the AOI concerning the RQ<sub>5</sub>. While there were substantial reductions in the fixation duration and fixations count (RQ<sub>4</sub>–RQ<sub>5</sub>), time and submissions were not so affected (RQ<sub>1</sub>–RQ<sub>2</sub>). In the obfuscated version, subjectively, confusion sources are more concentrated in the conditional expression. In the clarified version, they concentrate on variables and the condition.

### 5.4 Operator Precedence

In Fig. 20, we depict the obfuscating atom *Operator Precedence* on the left-hand side and the clarified version on the right-hand side. The only change in the clarified version consists of adding two parentheses to clarify the precedence of the boolean operators.

Knowing the order of precedence of the operators is essential to correctly solve the code since the wrong order yields a wrong output. For instance, (False and True) or True when correctly interpreted prints True, while False and (True or True),

<b>Obfuscated</b>	<b>Clarified</b>
if (False and True or True):	if ((False and True) or True):

Fig. 20 Obfuscating atom *Operator Precedence* and clarified version

evaluated wrongly, prints `False`. Concerning the  $RQ_1$ , we observed a reduction in the time spent in the AOI combined with fewer attempts ( $RQ_2$ ) with the clarified version. Those reductions were associated with reductions in the duration of the fixations ( $RQ_3$ ), fixations count ( $RQ_4$ ), and regressions count ( $RQ_5$ ). These results suggest that adding the parenthesis reduces the effort involved in comprehending the clarified version.

Eye tracking allows us to see the impact of adding the parenthesis at a fine-grained level. We observe that the clarified version reduced the median number of horizontal regressions by 28% in the code, possibly freeing the subjects from going back and forth in the statement trying to figure out the right order of the boolean operators. That becomes more emphasized in the reduction of regressions in the AOI by 47%.

Most subjects make a linear sequence of the three regions `FirstOperand` → `SecondOperand` → `ThirdOperand` in both versions. The regions are depicted in Fig. 21. However, in the obfuscated version, the subjects make 29% more transitions between these regions than the clarified one. When subjects reach the end of the entire expression, they should enter the true condition `ConditionTrue` as a result of correct comprehension. However, in the obfuscated version, this number of transitions is 77% smaller than in the clarified version, which can indicate that, due to the lack of understanding of priority, subjects tend to get back on the line to try to understand again. The number of regressions between `ThirdOperand` → `SecondOperand` is 52% higher in the obfuscated version, which may indicate a wrong association because of priority. In addition, in the obfuscated version, two subjects go from `ThirdOperand` → `ConditionFalse`, which is moving to the incorrect condition. Both only got the programs solved in the second attempt and presented time in the AOI above the average.

We isolated the subjects who submitted more than one answer to solve and compared the concentration of eye movements between `FirstOperand` → `SecondOperand` and `SecondOperand` → `ThirdOperand`. In the obfuscated version, `FirstOperand` → `SecondOperand` has the `and` operator which has precedence but no parenthesis. We found that the subjects make `FirstOperand` → `SecondOperand` 27 times while `SecondOperand` → `ThirdOperand` 28 times. They make `SecondOperand` → `FirstOperand` 15 times while `ThirdOperand` → `SecondOperand` 17 times. We

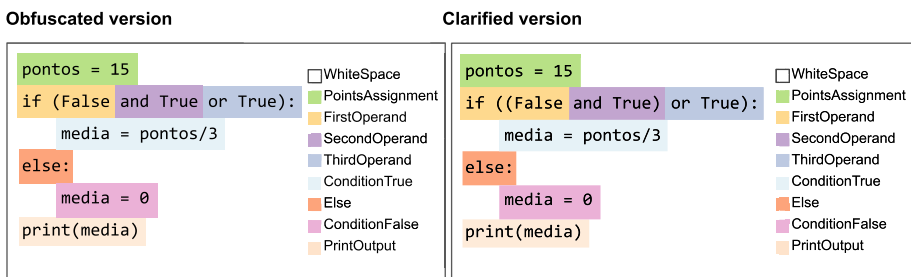


Fig. 21 Set of regions inside the code version with *Operator Precedence* atom and in the code with the clarified code version

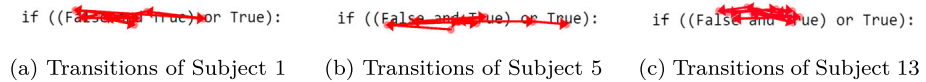


Fig. 22 Sequence of transitions of a subject on the clarified code version with *Operator Precedence*

learned that they go back and forth between these regions with a similar effort. However, they make `ThirdOperand`  $\rightarrow$  `FirstOperand` nine times and `FirstOperand`  $\rightarrow$  `ThirdOperand` three times, which might be an indication of confusion. The pattern `ThirdOperand`  $\rightarrow$  `FirstOperand` can indicate a wrong precedence involving the ‘or’ operator between the regions.

We performed a deeper analysis of the gaze transitions to understand the thought process of the subjects. For the clarified version of the *Operator Precedence*, the subjects exhibited transitions going forward and backward concentrating on the expression inside the parentheses. We observed the same transitions for other subjects. Consider the transitions exhibited by the following subjects in Fig. 22.

In the gaze, we observed transitions going forward and backward between ‘False and True’ which may indicate that the subject understands which sub-expression should be evaluated first. Subject 5 mentioned that it was easy to solve because of the parenthesis. On the other hand, for the obfuscated version of the *Operator Precedence*, we found transitions that indicate confusion. Consider the gaze transitions exhibited by the following subjects in Fig. 23.

In the gaze, we observed transitions going forward and backward repeatedly between the logical operators ‘and’ and ‘or’, which may indicate that the subjects were not certain about which operator should come first. Subject 2 mentioned having difficulties with the precedence, made an error in the first submission, and reported that only later on she realized that the ‘and’ had precedence over ‘or’. Consider the transitions exhibited by the subjects in Fig. 24.

We observed transitions going forward and backward between ‘True or True’ which may indicate that the subject had doubts about which expression should be evaluated first. Indeed, Subject 4 mentioned having difficulties with the ‘and’ and ‘or’ operator. The lack of parenthesis may lead to more transitions going forward and backward between the logical operators and lead to making wrong associations. Unlike the obfuscated version, in the clarified version we did not observe such transitions going forward and backward eye movements between ‘True) or True’.

The subjects perceived the obfuscated version as more difficult to solve. In the obfuscated version, the subjects mainly mentioned the following issues: “*order of precedence*”, “*if difficult*”, “*misunderstood if*”, “*if difficult to validate*”, “*misunderstood and with or*”, “*I missed parentheses*”, “*precedence*”, “*order confusing*”, “*boolean difficult*”, “*didn’t remember and or*”. In clarified version, they mentioned: “*confused true and false*”, “*validate and with or difficult*”, “*validation complicated*”. As a takeaway, in the obfuscated version,

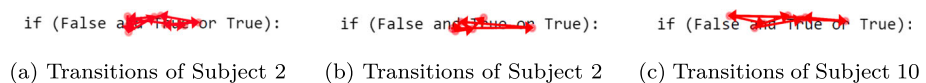


Fig. 23 Sequence of transitions of a subject on the obfuscated code version with *Operator Precedence*

sources of confusion are more concentrated in `if` condition, while in the clarified version, still in `if` condition, however, less often.

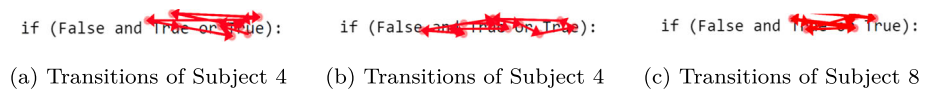
Answering our RQs, in the clarified version of code with *Operator Precedence*, there were reductions in the time in the AOI (RQ<sub>1</sub>), number of attempts (RQ<sub>2</sub>), duration of fixations (RQ<sub>3</sub>), fixations count (RQ<sub>4</sub>), and horizontal regressions count (RQ<sub>5</sub>). In the obfuscated version, subjectively, sources of confusion are more concentrated in `if` condition, while in the clarified version, still in `if` condition, however, less often.

## 5.5 Implicit Predicate

In Fig. 25, we depict the obfuscating atom *Implicit Predicate* on the left-hand side and the clarified version on the right-hand. The obfuscated version assumes that the expression in the condition of the `if` statement can be used as a predicate. The change in the clarified version consists of making the condition explicit by adding a comparison with zero. With the clarified version, we observed reductions in the time in the AOI (RQ<sub>1</sub>), number of attempts (RQ<sub>2</sub>), fixation duration (RQ<sub>3</sub>), fixations count (RQ<sub>4</sub>), and horizontal regressions count (RQ<sub>5</sub>).

In Fig. 26, we give an example of these reductions with two subjects who examine the program of SP<sub>1</sub> containing the *Implicit Predicate*, one subject in each version. The subject seems to go back more times in the code to decipher the missing information with the implicit predicate. The subjects report having difficulties in understanding the modulo in the obfuscated version.

In the obfuscated version, the subjects should evaluate the expression `IfFirstOperand` → `ModuloNumber` in Fig. 27 at most three times, once for each iteration in the loop. However, we observed that subjects make this transition considerably more times than necessary, which may indicate more effort to understand. It is possible to arrive at the result by performing the `ModuloNumber` → `IfFirstOperand` regression. However, the subjects regress more times than necessary. So we observe a frequent back-and-forth. Furthermore, in the obfuscated version, subjects look specifically at the region containing the modulo considerably more times than necessary, which might indicate difficulty with this region. In the clarified version, adding the predicate explicitly splits the reading effort between three regions: `IfFirstOperand`, `ModuloNumber`, and `ComparisonOperator`. With the explicit predicate, almost 50% of the subjects performed `IfFirstOperand` → `ModuloNumber` → `ComparisonOperator` linearly. The modulo region and `ComparisonOperator` are seen considerably more times than necessary. As a takeaway, both versions demonstrated diverse reading patterns, but the addition of the explicit predicate divides the effort by making it lower compared to the implicit predicate.



**Fig. 24** Sequence of transitions of a subject on the obfuscated code version with *Operator Precedence*

<b>Obfuscated</b>	<b>Clarified</b>
<code>if (elem % 5):</code>	<code>if (elem % 5 != 0):</code>

Fig. 25 Obfuscating atom *Implicit Predicate* and clarified version

In obfuscated version, the subjects mainly mentioned the following issues: “if is complicated”, “if is difficult”, “% symbol confusing”, “modulo”, “% made it difficult”, “difficulties with modulo”, “unsure about % part”, “% 5”, “didn’t understand %”, “if confusing”, “if hard to validate”, “difficult because of %”. In clarified version, they mentioned: “difficulties with % and !=”, “modulo”, “while and if together is difficult”, “incrementing”, “!= 0 confusing”, “counter in while difficult”, “confused %”, “if takes more time to understand”. As a takeaway, in the obfuscated version, the sources of confusion are more concentrated in the condition of the if statement, while in the clarified version, still in the if statement but with more diverse sources.

Answering our RQs, in the clarified version, we observe reductions in the time in the AOI (RQ<sub>1</sub>), number of attempts (RQ<sub>2</sub>), fixation duration (RQ<sub>3</sub>), fixations count (RQ<sub>4</sub>), and regressions count (RQ<sub>5</sub>). The highest impact was observed for the time in the AOI. In the obfuscated version, subjectively, the sources of confusion are more concentrated in the condition of the if statement involving the modulo operator, while in the clarified version, still in the if statement but with more diverse sources.

### 5.6 Augmented Operator

In Fig. 28, we depict the *Augmented Operator*. To make the expression shorter, the combination of the arithmetic operator with the assignment operator can confuse the subject about what receives the result of the operation. When the subjects have to pay more attention to the augmented operator or look more often at the assigned variable can give evidence of the effort in the understanding. With the clarified version, the subjects spent less time in the AOI (RQ<sub>1</sub>) and had a less visual effort with reductions in the fixation duration (RQ<sub>3</sub>), fixations count (RQ<sub>4</sub>), and regressions count (RQ<sub>5</sub>).

We observed that, in the obfuscated version, the region that contains the assignment symbol next to the operator with the assigned value, Assignment, is seen 37% more times than

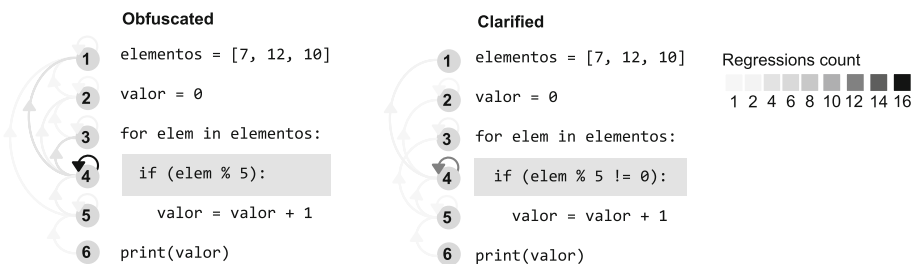


Fig. 26 Regressions graph with two subjects visually regressing horizontally and vertically in the code, one with the obfuscated code containing the *Implicit Predicate* and the other with the clarified version of the code

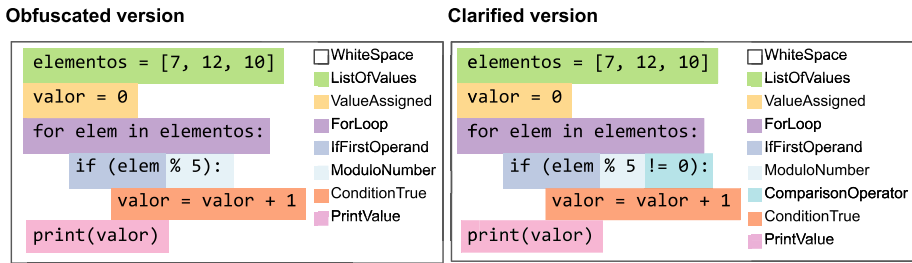


Fig. 27 Set of regions inside the code version with *Implicit Predicate* atom and in the code with the clarified version of code

the same region in the clarified. The regions are depicted in Fig. 29. The structure without the augmented operator may alleviate the effort of looking at the same variable every time on the same line.

The clarified version aims to clarify the operation for updating an integer variable by repeating the same variable even if it becomes more verbose. It is worth remembering that, in Python, we can update a variable by adding and assigning (`+=`), multiplying and assigning (`*=`), and using other operators. In our programs, we used these two operators. Thus, the clarified version has more elements to be observed by the subject. However, instead of increasing the visual effort, it is reduced, and the reduction in the number of regressions in the AOI is even more noticeable.

The subjects make fewer transitions between the AOI and the rest of the code in the clarified version containing the *Augmented Operator*. The subjects entered the AOI slightly fewer times with the clarified version than the obfuscated one. In Fig. 30, we give an example of these reductions. In the obfuscated version, in the example, the transitions between the AOI and the rest of the code are more intense than in the clarified version. Most of the difference in the number of transitions occurs between AOI and the upper part of the code. Since we have a loop that iterates over three elements, the number of entries and exits should be three. It seems that using syntactic sugar in the obfuscated version leads the subjects to turn to the upper part more times, at least for the multiply and assign operator, as reported by the subjects.

In obfuscated version, the subjects mainly mentioned the following issues: “`*=` symbol is strange”, “element variable is confusing”, “didn’t recognize `*=` symbol”, “for and elem are difficult”. In clarified version, they mentioned: “for and elem”, “confused values”, “difficulties with elem variable”, “didn’t understand the list”, “lost myself in the values”. As a takeaway, in the obfuscated version, the sources of confusion are more concentrated in the lack of knowledge of the `*=` symbol, while in the clarified version, the sources were concentrated in the values.



Fig. 28 Obfuscating atom *Augmented Operator* and clarified version

Answering our RQs, in the clarified version, we observed reductions in the time spent in the AOI (RQ<sub>1</sub>) as well as in the fixation duration (RQ<sub>3</sub>), fixations count (RQ<sub>4</sub>), and regressions count (RQ<sub>5</sub>). The number of attempts slightly increased (R<sub>2</sub>). In the obfuscated version, subjectively, the sources of confusion are more concentrated in the lack of knowledge of the \*= symbol, while in the clarified version, the sources were concentrated in the values.

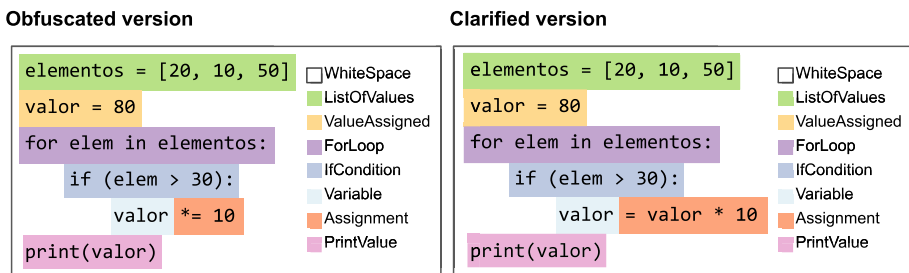
### 5.7 Coding Subjects' Answers

We used the method of the grounded theory proposed by Strauss and Corbin (1998) to analyze our qualitative data. A tentative explanation for most of the quantitative results is the presence of certain obfuscating atoms. However, by employing the grounded theory, we aim to understand and discuss whether we have qualitative evidence to support this theory in our study or whether the qualitative evidence reveals other alternative potential sources of confusion.

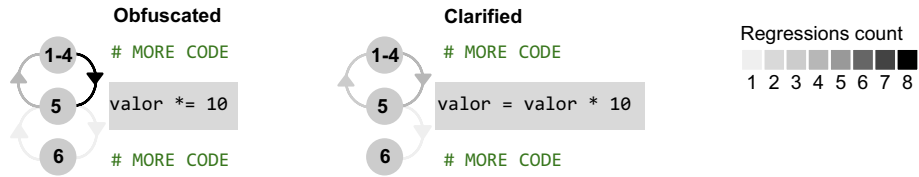
The grounded theory aims at coding and categorizing to describe a phenomenon found in the data, avoiding preconceived theories to focus on only the data. We used the following steps. First, during the interview, we ask questions to the subjects, break their answers into smaller chunks of data, and identify the major idea by assigning it a code that emerged from their answers. Thus, we perform coding in the first step. Second, we read all the codes and search for opportunities to group them into higher-level concepts. Third, we identify categories by discussing how similar the concepts were according to their properties. Fourth, we derive a theory through an inductive approach. All these steps can be seen in Table 3. We make all these steps available with more detail in our replication package on Zenodo (da Costa et al 2023).

In the code step, we coded the answers of the subjects. We focused on synthesizing what phenomenon is described by the subject. Our codes ranged from a single word to short sequences of words. We had a set of 142 codes across 384 answers. Our codes included issues such as “*I was confused with if ternary*”, “*confusion with the rest of the division*”, and “*trouble in memorizing the variables' assignments*”. More details can be seen in Table 3.

In the concept step, we focused on abstracting, connecting, and grouping multiple codes. We found 22 concepts that emerged from the codes. For instance, we grouped “trouble in



**Fig. 29** Set of regions inside the code version with *Augmented Operator* atom and in the code with the clarified version of code



**Fig. 30** Transitions graph with two subjects visually entering and exiting the AOI in the code, one with the obfuscated code containing the *Augmented Operator* and the other with the clarified version of the code

memorizing the variables’ assignments” and “it was difficult to remember the value of the variable” in the same underlying concept, which is “memorization of values”.

In the category step, we group the concepts into more abstract categories. Our concepts were included in seven broad categories, namely code style, control/repetition structures, knowledge, mathematics, memory load, operators, and no difficulties. Code style comprised concepts such as indentation or multiple assignments in the same line; control flow comprised

**Table 3** Steps of the coding process of the subjects’ answers

Code Step	Concepts Step	Category Step
Many assignments in one line complicates	Single line assignments	Code Style
Got confused in indentation	Indentation	
<code>if</code> inside loop is difficult	nesting structures	Control Flow
Evaluating <code>if</code> takes longer	Conditional structure	
Few iterations over loop	Repetition structure	
Not used to ternary	Knowledge of Idiom	Knowledge
<code>for</code> in Python is strange	Knowledge of Language	
Simple calculations	Math calculation	Mathematics
Difficulties with the division	division calculation	
Incrementing is difficult	Increment	Memory Load
I got lost in the iteration	Iteration	
Index of the loop confuses	Index of loop	
Trouble in memorizing the variables	Memorization of Values	
Too many variables	Amount of Variables	
Difficulties in swapping the variables	Swapping Variables	
Unnecessary variable	Temporary Variable	
Difficulties with modulo and arithmetic operators	Combination of Operators	Operators
Found modulo operator confusing	Arithmetic Operator	
Found boolean operator complex	Relational Operator	
Confused <code>true</code> and <code>false</code>	Logical Operators	
Difficulties with Precedence	Operator Precedence	
I had no difficulties	No difficulties	No difficulties



both conditional structures, repetition structures, and their combination nested; knowledge comprised both knowledge of programming language and idiom. More details can be seen in Table 3.

A research question emerged from the data which was: what are the potential sources of confusion found in the data? From the categories, we derived a theory describing the factors that influence the understanding of the subjects in the study, which can provide an understanding of the sources of confusion. For instance, the theory revealed that sources of confusion were diversified in the context of novices in Python involving aspects of the code such as the style and structure, and external factors such as knowledge domain and memory load.

Regarding the flow category, we found 31 codes related to the three concepts: conditional structures, repetition structures, and their combination. Most of the atoms we evaluated have control flow which can explain why this category is so broad. However, the theory revealed that difficulties were associated more frequently with the obfuscated versions of the code in this category, especially in the atom *Conditional Operator*. For instance, we found six codes associating its obfuscated version with more complication, confusion, difficulties, and more time. Even though it did not affect the number of attempts, three subjects mentioned “*It takes longer to validate the if statement*”. Indeed, we observed that the subjects fixated for a longer duration in the AOI, which affects time, and presented more horizontal regressions.

Regarding the memory load category, we found 36 codes related to seven concepts, comprising memorization of values, iterating over loops, and swapping variables, among others. We carefully designed the tasks to have a few variables and iterate over a short list to avoid memory load effects. The clarified version of *Multiple Variable Assignments* was the main atom related to the concepts associated with the number of variables. The clarified version repeats one variable and breaks the assignment into two lines. The subjects mentioned that “*There are too many variables*” and we observed more visual horizontal regressions associated with these specific programs. We did not observe an impact in the number of attempts, but the clarified version might indicate that short-term memory can be affected.

Regarding the operators’ category, we found 29 codes related to five concepts comprising arithmetic, relational, and logical operators, their combination, and precedence. The obfuscating atom *Operator Precedence* did not make explicit in which order the subjects should evaluate the expression, and we had 21 answers related to difficulties in identifying the correct order or with the easiness in using parenthesis for that purpose. Six subjects need two attempts to solve the programs and present more horizontal regressions. One subject that needs one more attempt mentioned “*I evaluated from left to right, then I realized that I should examine the operator AND first*”.

## 6 Threats to Validity

Here we describe potential issues and threats to the validity of our study: internal validity (Section 6.1), external validity (Section 6.2), and construct validity (Section 6.3).

### 6.1 Internal Validity

We performed the experiment in four locations to gather more subjects and have a variety of subjects from distinct higher-education institutions. However, different locations may influence the visual attention of the subjects. To mitigate this, we carefully arranged the rooms

to have similar conditions. For instance, they were quiet rooms with minimum distractions, similar temperatures, and artificial light sources. In future work, we aim to keep track of which subject performed over which location so we can bind possible differences.

Despite our best efforts, the presence of a researcher in the room may have unintentionally influenced the visual attention or performance of the subjects since they were aware of being observed. To mitigate this threat, we put effort into letting the subjects feel comfortable with the researcher's presence. In addition, we avoided any interaction with the subjects while they were examining the programs so that they could be concentrated.

Given the camera limitations, we needed to adjust the gaze points for some subjects. The adjustment in the points influences the interpretation. However, the authors discussed the adjustments by going systematically through the data for each subject. Thus, to mitigate the threat of working with uncalibrated equipment we generated another threat. However, the authors decided that the threat of adjusting the points would be preferable to analyze the data with points not touching the code, given the uncalibrated equipment. It is important to mention that the median number of pixels used to correct the fixations in  $y$ -coordinate was 30 pixels, which translated to 0.6 lines of inaccuracy on the screen, and the maximum value was 80 pixels. For the  $x$ -coordinate, following this strategy, we did not need to adjust the  $x$ -coordinate. We made the fixations and adjustment strategy available in our replication package (da Costa et al 2023).

In pilot studies, we observed that a swivel chair could impair data collection by the camera or negatively affect the captured data. To mitigate this threat, we used chairs without swiveling capability in all rooms we used to conduct the experiment.

The total time we allocated for each subject was one hour, and we assigned them 12 programs, which may have influenced the visual effort. To minimize this threat, we designed simple and short programs with only one atom instantiated and put a time limit of two minutes. Given the simplicity of the programs, most subjects solved them before the time limit. Since our programs consist of non-minimal snippets, in the sense that they do not contain only the atom region, the extra lines of the code might introduce working memory as a confounding factor. However, with eye tracking, it is possible to measure and compare time and visual effort only in the atom region. In addition, both programs, in obfuscated and clarified versions with the same atom, had the same extra lines of code to make the comparison fair.

All the subjects had the option to keep making attempts until getting the correct output. We then compared the number of attempts until they answered correctly. However, following this strategy, if one makes wrong attempts, she could make more fixations or even longer ones, with more regressions. Alternatively, we collected the eye tracking data separately for each attempt made to compare only the first one. We performed an analysis based only on the data from the first attempt, but we found similar results.

Using the Latin Square design, we blocked the set of programs to control noise. Besides performing combinations of the programs in the squares, we analyzed the programs with the atoms individually. The analysis of individual programs in the set of programs violates the design. The extent of such violation does not have an estimated impact. However, to better understand the effects of the atoms, analyzing them combined and individually can give a more nuanced and complete understanding of their effects.

## 6.2 External Validity

We resorted to small programs with less than 10 lines of code aiming at fitting the code onto the screen. This approach may restrict generalization to larger programs. However,

previous work on the same subject has resorted to code snippets with a similar number of lines (Gopstein et al. 2017; Oliveira et al 2020). If we find differences in small code snippets, we expect that larger snippets may tend to show greater differences. Nevertheless, we need to conduct other studies with larger code snippets to provide empirical evidence for those expectations.

In our study, we focused on novices in Python. Thus, we cannot generalize to more experienced developers in Python. Novices have also been the subject of other eye tracking studies on code comprehension (Busjahn et al 2015). In the future, we intend to explore the same topic of this study with experienced developers.

Since we have focused on atoms in Python programming language, we cannot generalize our findings to other programming languages. To mitigate this threat, in our programs, we used constructions common in other languages, and most of our subjects reported some experience with other languages. In addition, since our subjects were Brazilian Portuguese native speakers, our programs were designed to contain identifiers in their mother tongue.

Our programs were designed to have only one output, a numeric value. All subjects had to solve the task by specifying the correct output aloud after reading the code. The results for this type of task may not generalize to other types, such as finding a bug, fixing a syntax problem, or adding a feature. In addition, since the font style may influence the subject's attention, to minimize a possible threat, we consistently used the same font style and size for the programs, with no syntax highlighting and no bold font.

The number of atoms instantiated in a program may influence the performance and visual effort of the subjects. To minimize threats related to the number of atoms, we consistently used only one atom in each program.

### 6.3 Construct Validity

Time and answer correctness are often employed to assess code comprehension (Schulze et al 2013; Malaquias et al 2017) and in particular, to investigate atoms of confusion (Gopstein et al. 2017; de Oliveira et al 2020). Concerning eye tracking methodology, other studies have employed similar metrics to measure visual-related aspects (Melo et al 2017; Sharif and Maletic 2010; Bednarik and Tukiainen 2006). Other works have combined time, answer correctness, and visual effort (Sharif et al 2012; de Oliveira et al 2020; da Costa et al 2021). In particular, fixation duration and fixations count have been used to measure visual effort (Sharif et al 2012; Binkley et al 2013). According to Sharafi et al. (2015), metrics based on saccades, such as the number of saccades or saccades duration, are metrics whose definitions are identical to the ones based on fixations. Thus, we decided to explore eye movement regressions since they have been explored and associated with visual effort (Sharafi et al 2015).

Inviting people to participate in eye tracking studies may influence the subjects' decisions regarding their visual behavior. For instance, we have to make them aware that their eyes are being tracked, which may influence where or how much they look at some regions of the code. To minimize this threat, we did not make the subjects aware of the precise goals of the study to avoid hypothesis guessing.

## 7 Related Work

In this section, we present the related work. We present works related to the atoms of confusion (Section 7.1), code comprehension (Section 7.2), and eye tracking experiments (Section 7.3).

## 7.1 Atoms of Confusion

Gopstein et al. (2017) introduced the term “atom of confusion” as the smallest code pattern that can cause misunderstanding in the programmer. They proposed a set of 15 atoms they extracted from the International Obfuscated C Code Contest. They hypothesized that these atoms could cause programmers to misunderstand code. They performed two empirical experiments, one with 73 subjects and the other with 43 mostly students, aiming to find which atoms caused confusion and how much confusion they could reduce by clarifying the atoms. They measured the time it took for programmers to answer correctly and the accuracy of their answers. They found that small C code snippets, including atoms of confusion, are more difficult to understand than their functionally equivalent clarified versions. Extending their prior work, Gopstein et al. (2018) investigated the prevalence of atoms of confusion in the real-world setting. They performed a study involving 14 open-source projects in the C language and found that atoms of confusion are prevalent in real and successful projects. In addition, the presence of these atoms correlates with bug-fixing commits and long code comments. We performed a controlled experiment to observe the impact of the obfuscated code containing atoms of confusion and the clarified code on the novices’ code comprehension. However, we focused on Python programming language and, besides time and accuracy, we investigated the eye tracking metrics. In addition, we were more conservative in our programs than in their studies, using more meaningful names for the variables. This approach is arguably closer to a practical scenario.

Gopstein et al. (2020) performed a study with 14 human subjects, including both professionals and students, aiming to understand better and scrutinize their prior studies on atoms of confusion. According to them, precision and accuracy can only tell the outcome of programmers’ performance, but not how or why programmers behaved in a certain way. They used a think-aloud methodology to collect data and then performed a qualitative analysis. They found that correct hand-evaluations do not imply understanding, which means that a subject can answer correctly and still be confused. Similarly, incorrect evaluations do not imply misunderstanding. With the sole use of accuracy, these sources of confusion would otherwise go unnoticed. Going beyond accuracy, we used an eye tracker to assess the visual effort of the subjects. Eye tracking allowed us to understand their visual behavior better while solving the tasks, which could give insights into how or why programmers behaved in a certain way. In addition, we also performed a qualitative interview to get personal feedback.

Medeiros et al. (2019) aimed to understand the relevance and prevalence of atoms of confusion in C open-source projects. They used a mixed research method approach, which comprised mining repositories of 50 C open-source projects and a survey with 97 developers with experience in the C language. They found that atoms of confusion are prevalent in open-source projects, with more than 109K occurrences of the 12 atoms. In addition, according to developers’ opinions, only some atoms are perceived to cause misunderstandings. Instead of basing on the opinions of experienced developers in C, we conducted a controlled experiment to quantitatively and qualitatively assess the performance of the human subjects who were novices in Python. In addition to time and accuracy, we collect eye tracking metrics to have a better understanding of the effect of the atoms. To complement our quantitative data, additionally, we perform interviews to get feedback from the subjects.

## 7.2 Code Comprehension

Yeh et al. (2021) used an EEG device to measure the cognitive load of the developers as they attempted to predict the output of C code snippets. They aimed to observe whether particular

patterns within the code snippet induced higher levels of cognitive load. They found that particular patterns indeed affect the developers' cognitive processes. We focused on eye movements instead of brain activity. We explored in more depth the visual effort regarding atoms in Python language.

Langhout and Aniche (Langhout and Aniche 2021) replicated the work of Gopstein et al. (2017), however, in the Java programming language. After deriving a set of atoms of confusion for Java, they performed an experiment with 132 novices. They found that atoms of confusion can cause confusion among novice developers. Extending this idea, Mendes et al. (2021) proposed a tool named BOHR (The Atoms of Confusion Hunter) to detect atoms of confusion in Java systems. The tool detected eight out of 13 types of atoms pointed out as confusing by Langhout and Aniche (2021). We also investigated the potential of atoms to influence the code comprehension of novices negatively. However, we did so in Python language and from the perspective of the eye tracking measures.

Castor (2018) proposed a structured definition of atoms of confusion, examined factors that make them confusing, and presented a preliminary catalog of atoms of confusion for the Swift programming language. Based on the prior studies (Gopstein et al. 2017; Gopstein et al 2018), Castor defined an atom as precisely identifiable, likely to cause confusion, replaceable by another pattern that is less like to cause confusion, and indivisible. He also identified sources that make atoms confusing, such as little-known and less common constructs, which include *Conditional Operator* and *Assignment as Value*. We used the definition already proposed (Gopstein et al. 2017; Gopstein et al 2018) and empirically investigated the effects of obfuscated and clarified programs on code comprehension in Python language.

Schröter et al. (2017) conducted a literature review to investigate how researchers address code comprehension. Among their findings, they found that the source code and program behavior are the mostly addressed parts of code comprehension in their empirical studies. Our work consists of an empirical study that compares programs following distinct styles.

### 7.3 Eye Tracking

Oliveira et al. (2020) studied the impact of atoms on time, accuracy, and the subjects' focus of attention with an eye tracker. They conducted a controlled experiment with students and software practitioners. They evaluated code snippets from real open-source C/C++ systems of different domains containing three types of atoms, *Assignment as Value*, *Conditional Operator*, and *Logic as Control Flow*. They found differences in time and visual attention of the subjects. However, they did not explore fixation duration, fixations count, and regressions count. Besides using these visual metrics, we evaluated more atoms instantiated in code snippets in Python language.

Sharafi et al. (2010) studied the influence of the styles camel case and underscore on code comprehension. They measured time, answer correctness, and visual effort through eye tracking. They found a significant improvement in time and visual effort with the underscore style. Sharafi et al. (2012) investigated the impact of the same two styles on code comprehension by considering the gender of the subjects. Measuring time, accuracy, and visual effort, no differences were observed. In our study, we considered similar metrics—time, number of attempts, and visual effort—however a different context.

Stefik and Siebert (2013) studied the influence of programming language syntax on the novices' comprehension. As their tasks, the novices had to rate the intuitiveness of several programming language constructs. Among the findings of the study, syntactic choices made in commercial programming languages are more intuitive to novices than others, and variations

in syntax influence novice accuracy rates when they are starting to program. We also explored the context of novices. However, we considered just one language and used objective metrics while novices solved the code task.

Obaidellah et al. (2018) conducted a systematic mapping study on eye tracking experiments focusing on code scenario. They found that the main areas of research include program comprehension and debugging, non-code comprehension, collaborative programming, and requirements traceability research. In addition, they found that most of the subjects in the experiments were students and faculty members from institutions. In our controlled experiment, we focus on code comprehension involving novices.

## 8 Conclusions

In this article, we reported on eye tracking as a method to gain new insights into the atoms of confusion. We performed a controlled experiment with eye tracking to evaluate the impact of six atoms of confusion on code comprehension. We evaluated to what extent the obfuscated code containing atoms of confusion and the functionally equivalent clarified versions of the code impacted the time, number of attempts, and visual effort of 32 novices in Python.

With eye tracking, we investigated how much time the subjects spent in a specific region of the code that contained the atoms of confusion and their clarifying versions, to what extent the atoms impacted the fixation duration, fixations count, and regressions count, and how the atoms impacted the way the subjects read the code. Thus, our findings contribute with some relevant implications. For the education community, our study contributes to raising concerns regarding teaching methods that may hinder code comprehension for Python novices. Educators should be careful when preparing the teaching material for introductory courses, avoiding using code snippets with atoms that can confuse the novices. For instance, the subjects in our study needed 28.3% more attempts to solve the code containing the *Operator Precedence*, which is associated with a negative impact on their abilities to understand the code. For Python novices, the positive impact of most of the clarified versions of the code containing the atoms in the time in the AOI and fixation duration, fixations count, and regressions count may indicate improvements in their productivity, understanding, and visual effort.

For the research community, our study setup exploring the visual effort dimension contributes to nuances not observed by previous works. For instance, in the analysis of the visual data for code containing the *Multiple Variable Assignment*, we perceived that the use of multiple assignments within the same line impacted the way the subjects read the code. The code with *Multiple Variable Assignment* allowed the subjects to read the assignments in a more direct manner, with 60% fewer regressions in the AOI. When the assignments are split between two lines, to make the code clearer, the subjects tended to make more vertical regressions and to keep coming back to those lines, transitioning between those lines and the lines of code that later use them. Hopefully, this will encourage researchers to consider eye tracking as a promising alternative to evaluate atoms of confusion. Other dimensions, such as mapping neural activities with Functional Magnetic Resonance Imaging (fMRI) or tracking all the subjects' activity during the experiments, could possibly reveal other nuances and allow us to dive deeper into how this atom impacts difficulty beyond visual effort. This can be a future direction for research. For practitioners and language designers, the use of syntactic sugar in the language syntax has to be done considering whether the pattern will impair the novices' abilities to understand the code. Some languages have abolished constructs because they can create obstacles for novices. For example, C-style `for` loops were removed from

Swift (Sadun and Gregor 2015) because, among other things, they offer “*a steep learning curve from users arriving from non C-like languages*”.

In future work, we aim to evaluate other types of atoms proposed by Gopstein et al. (2017). Cedrim et al. (2017) studied the influence of refactorings on code smells and found that their majority are neutral, and some refactorings even lead to new smells in the code. We aim to explore this topic with the perspective of eye tracking. In addition, we aim at conducting more experiments with experienced developers in Python, with a larger number of subjects, explore other programming languages, programs with variables names that have no meaning at all, other types of tasks such as finding a bug and investigate a higher number of atoms instantiated in a single task. Finally, we intend to add other eye tracking metrics based on saccades, blink rate, and pupil dilation and explore code reading patterns based on gaze transitions. As future work, we envision the development of more advanced tools that track the eye movements of developers and assist them with tips. For instance, if a tool detects transitions going forward and backward eye movements between two operators such as ‘and’ and ‘or’, or between an expression that has no precedence, a tip should arise to add the parentheses. We also envision proposing heuristics or building a model whereby a programmer receives an arbitrary source code, and we use eye tracking data to identify which elements were atom candidates or infer confusing regions that negatively impacted the code comprehension of the programmer.

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

**Data Availability** The dataset generated during the current study is available on Zenodo repository within a replication package also containing forms, programs, fixation data, data correction strategy, and other materials (da Costa et al 2023).

## Declarations

**Conflict of Interest** The authors declared that they have no conflict of interest.

## References

- Basili V, Caldiera G, Rombach H (1994) The Goal Question Metric Approach. *Encycl Softw Eng* 2:528–532
- Bednarik R, Tukiainen M (2006) An Eye-tracking Methodology for Characterizing Program Cprehension Processes. In: Proceedings of the Symposium on Eye Tracking Research & Applications, ETRA’06, pp 125–132. Association for Computing Machinery, New York
- Binkley D, Davis M, Lawrie D, Maletic J, Morrell C, Sharif B (2013) The Impact of Identifier Style on Effort and Comprehension. *Empir Softw Eng* 18(2):219–276
- Box G, Hunter JS, Hunter WG (2005) *Statistics for Experimenters*. Wiley-Interscience
- Busjahn T, Bednarik R, Begel A, Crosby M, Paterson JH, Schulte C, Sharif B, Tamm S (2015) Eye Movements in Code Reading: Relaxing the Linear Order. In: Proceedings of the International Conference on Program Comprehension, ICPC’15, IEEE, pp 255–265
- Busjahn T, Schulte C, Busjahn A (2011) Analysis of Code Reading to Gain More Insight in Program Comprehension. In: Proceedings of the Koli Calling International Conference on Computing Education Research, Koli Calling’11, pp 1–9. Association for Computing Machinery, New York
- Castor F (2018) Identifying Confusing Code in Swift Programs. In: Proceedings of the CBSoft Workshop on Visualization, Evolution, and Maintenance, VEM’18. ACM, São Carlos
- Cedrim D, Garcia A, Mongiovi M, Gheyi R, Sousa L, de Mello R, Fonseca B, Ribeiro M, Chávez A (2017) Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: Proceedings of the Joint Meeting on Foundations of Software Engineering, ESEC/FSE’17, ACM, pp 465–475

- Crosby M, Scholtz J, Wiedenbeck S (2002) The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In: Workshop of the Psychology of Programming Interest Group, PPIG'02, p 5. Brunel University
- da Costa JAS, Gheyi R, Castor F, Roberto P, Ribeiro M, Fonseca B (2023). Seeing Confusion Through a New Lens: On the Impact of Atoms of Confusion on Novices' Code Comprehension (Artifacts). <https://doi.org/10.5281/zenodo.7650076>
- da Costa JAS, Gheyi R, Ribeiro M, Apel S, Alves V, Fonseca B, Medeiros F, Garcia A (2021) Evaluating Refactorings for Disciplining `#ifdef` Annotations: An Eye Tracking Study with Novices. *Empir Softw Eng* 26(5):1–35
- de Oliveira B, Ribeiro M, da Costa JAS, Gheyi R, Amaral G, de Mello R, Oliveira A, Garcia A, Bonifácio R, Fonseca B (2020) Atoms of Confusion: The Eyes Do Not Lie. In: Proceedings of the Brazilian Symposium on Software Engineering, SBES'20, pp 243–252
- Galley N, Betz D, Biniossek C (2015) Fixation Durations: Why are They so Highly Variable. *Adv Vis Percept Res* 93:83–106
- Gopstein D, Fayard A-L, Apel S, Cappos J (2020) Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion. In: Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'20, pp 605–616. Association for Computing Machinery, New York
- Gopstein D, Iannacone J, Yan Y, DeLong L, Zhuang Y, Yeh MK-C, Cappos J (2017) Understanding Misunderstandings in Source Code. In: Proceedings of the Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, pp 129–139. Association for Computing Machinery, New York
- Gopstein D, Zhou HH, Frankl P, Cappos J (2018) Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In: Proceedings of the International Conference on Mining Software Repositories, ICSMR'18, pp 281–291. Association for Computing Machinery, New York
- Holmqvist K, Nyström M, Andersson R, Dewhurst R, Jarodzka H, Van de Weijer J (2011) *Eye Tracking: A Comprehensive Guide to Methods and Measures*. OUP Oxford
- Jadhav A, Pramod D, Ramanathan K (2019) Comparison of Performance of Data Imputation Methods for Numeric Dataset. *Appl Artif Intell* 33(10):913–933
- Just MA, Carpenter PA (1980) A Theory of Reading: From Eye Fixations to Comprehension. *Psychol Rev* 87(4):329
- Kleinke K (2017) Multiple Imputation under Violated Distributional Assumptions: A Systematic Evaluation of the Assumed Robustness of Predictive Mean Matching. *J Educ Behav Stat* 42(4):371–404
- Langhout C, Aniche M (2021) Atoms of Confusion in Java. In: Proceedings of the International Conference on Program Comprehension, ICPC'21, IEEE, pp 25–35
- Lawrie D, Feild H, Binkley D (2007) Quantifying Identifier Quality: an Analysis of Trends. *Empir Softw Eng* 12(4):359–388
- Malaquias R, Ribeiro M, Bonifácio R, Monteiro E, Medeiros F, Garcia A, Gheyi R (2017) The Discipline of Preprocessor-Based Annotations – Does `#ifdef` TAG n't `#endif` Matter. In: Proceedings of the International Conference on Program Comprehension, ICPC'17, IEEE, pp 297–307
- Medeiros F, Lima G, Amaral G, Apel S, Kästner C, Ribeiro M, Gheyi R (2019) An Investigation of Misunderstanding Code Patterns in C Open-source Software Projects. *Empir Softw Eng* 24(4):1693–1726
- Melo J, Narcizo FB, Hansen DW, Brabrand C, Wasowski A (2017) Variability Through the Eyes of the Programmer. In: Proceedings of the International Conference on Program Comprehension, ICPC'17, IEEE Press, pp 34–44
- Mendes W, Viana W, Rocha L (2021) BOHR - Uma Ferramenta para a Identificação de Átomos de Confusão em Códigos Java. In: Workshop de Visualização, Evolução e Manutenção de Software, VEM'21, SBC, pp 41–45. Sociedade Brasileira de Computação
- Nyström M, Holmqvist K (2010) An Adaptive Algorithm for Fixation, Saccade, and Glissade Detection in Eyetracking Data. *Behav Res Methods* 42(1):188–204
- Obaidallah U, Al Haek M, Cheng PC-H (2018) A Survey on the Usage of Eye-tracking in Computer Programming. *ACM Comput Surv (CSUR)* 51(1):1–58
- Oliveira D, Bruno R, Madeiral F, Castor F (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. Association for Computing Machinery, New York
- Rayner K (1998) Eye Movements in Reading and Information Processing: 20 Years of Research. *Psychol Bull* 124(3):372
- Sadun E, Gregor D (2015) Remove c-style for-loops with conditions and incremeters. Swift Programming Language Evolution, proposal SE-0007. <https://github.com/apple/swift-evolution/blob/main/proposals/0007-remove-c-style-for-loops.md>. Accessed 21 March 2023




- Salvucci D, Goldberg J (2000) Identifying Fixations and Saccades in Eye-tracking Protocols. In: Proceedings of the Symposium on Eye Tracking Research & Applications, ETRA'00, pp 71–78. Association for Computing Machinery, New York
- Schröter I, Krüger J, Siegmund J, Leich T (2017) Comprehending Studies on Program Comprehension. In: Proceedings of the International Conference on Program Comprehension, ICPC'20, IEEE, pp 308–311
- Schulze S, Liebig J, Siegmund J, Apel S (2013) Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. In: Proceedings of the International Conference on Generative Programming: Concepts & Experiences, GPCE '13, pp 65–74. Association for Computing Machinery, New York
- Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). *Biometrika* 52(3/4):591–611
- Sharafi Z, Shaffer T, Sharif B, Guéhéneuc Y-G (2015) Eye-tracking Metrics in Software Engineering. In: Proceedings of the Asia-Pacific Software Engineering Conference, APSEC'15, IEEE, pp 96–103
- Sharafi Z, Sharif B, Guéhéneuc Y-G, Begel A, Bednarik R, Crosby M (2020) A Practical Guide on Conducting Eye Tracking Studies in Software Engineering. *Empir Softw Eng* 25(5):3128–3174
- Sharafi Z, Soh Z, Guéhéneuc Y-G, Antoniol G (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, pp 27–36
- Sharif B, Falcone M, Maletic J (2012) An Eye-tracking Study on the Role of Scan Time in Finding Source Code Defects. In: Proceedings of the Symposium on Eye Tracking Research & Applications, ETRA'12, ACM, pp 381–384
- Sharif B, Maletic J (2010) An Eye Tracking Study on Camelcase and Under\_score Identifier Styles. In: Proceedings of the International Conference on Program Comprehension, ICPC'10, IEEE, pp 196–205
- Sheskin DJ (2020) Handbook of Parametric and Nonparametric Statistical Procedures. CRC Press
- Stefik A, Siebert S (2013) An Empirical Investigation into Programming Language Syntax. *Trans Comput Educ* 13(4):1–40
- Strauss A, Corbin J (1998) Basics of Qualitative Research Techniques. Citeseer
- Yeh MK-C, Yan Y, Zhuang Y, DeLong LA (2021) Identifying Program Confusion Using Electroencephalogram Measurements. *Behav Inf Technol* 41:1–18

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

## Authors and Affiliations

José Aldo Silva da Costa<sup>1</sup>  · Rohit Gheyi<sup>1</sup> · Fernando Castor<sup>2</sup> ·  
Pablo Roberto Fernandes de Oliveira<sup>1</sup> · Márcio Ribeiro<sup>3</sup> · Baldoino Fonseca<sup>3</sup>

Rohit Gheyi  
rohit@dsc.ufcg.edu.br

Fernando Castor  
f.j.castordelimafilho@uu.nl

Pablo Roberto Fernandes de Oliveira  
pablo@copin.ufcg.edu.br

Márcio Ribeiro  
marcio@ic.ufal.br

Baldoino Fonseca  
baldoino@ic.ufal.br

<sup>1</sup> Federal University of Campina Grande, Campina Grande, Brazil

<sup>2</sup> Utrecht University, Utrecht, Netherlands

<sup>3</sup> Federal University of Alagoas, Maceió, Brazil