UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ JÚNIOR SILVA DA COSTA

EVALUATING PYTHON REPETITION STRUCTURES WITH NOVICES: AN EYE TRACKING STUDY

CAMPINA GRANDE - PB
2024

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Evaluating Python Repetition Structures with Novices: an eye tracking study

José Júnior Silva da Costa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Rohit Gheyi

(Orientador)

Campina Grande, Paraíba, Brasil

C837e Costa, José Júnior Silva da.

Evaluating Python repetition structures with novices: an eye tracking study / José Júnior Silva da Costa. – Campina Grande, 2024.

74 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2024.

"Orientação: Prof. Dr. Rohit Gheyi".

Referências.

1. Engenharia de Software. 2. Code Comprehension. 3. Eye Tracking, 4. Repetition Structures. I. Gheyi, Rohit. II. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

REGISTRO DE PRESENÇA E ASSINATURAS

ATA Nº 011/2024 (DISSERTAÇÃO N° 725)

Aos oito (8) dias do mês de março do ano de dois mil e vinte e quatro (2024), às dezesseis horas (16:00), de forma remota, através da plataforma do GOOGLE MEET, reuniu-se a Comissão Examinadora composta pelos Professores ROHIT GHEYI, Dr., UFCG, Orientador, funcionando neste ato como Presidente, MÁRCIO DE MEDEIROS RIBEIRO, Dr., UFAL, IVAN DO CARMO MACHADO, Dr., UFBA. Constituída a mencionada Comissão Examinadora pela Portaria Nº 010/2024 da Coordenação do Programa de Pós-Graduação em Ciência da Computação, tendo em vista a deliberação do Colegiado do Curso, tomada em reunião de 15 de Fevereiro de 2024 e com fundamento no Regulamento Geral dos Cursos de Pós-Graduação da Universidade Federal de Campina Grande - UFCG, juntamente com o Sr(a) JOSÉ JÚNIOR SILVA DA COSTA, candidato(a) ao grau de MESTRE em Ciência da Computação, presentes ainda professores e alunos do referido centro e demais presentes. Abertos os trabalhos, o(a) Senhor(a) Presidente da Comissão Examinadora anunciou que a reunião tinha por finalidade a apresentação e julgamento da dissertação "EVALUATING PYTHON REPETITION STRUCTURES WITH NOVICES: AN EYE TRACKING STUDY", elaborada pelo(a) candidato(a) acima designado, sob a orientação do(s) Professor(es) ROHIT GHEYI, com o objetivo de atender as exigências do Regulamento Geral dos Cursos de Pós-Graduação da Universidade Federal de Campina Grande - UFCG. A seguir, concedeu a palavra, ao (a) candidato(a), o qual, após salientar a importância do assunto desenvolvido, defendeu o conteúdo da dissertação. Concluída a exposição e defesa do(a) candidato(a), passou cada membro da Comissão Examinadora a arguir o(a) mestrando sobre os vários aspectos que constituíram o campo de estudo tratado na referida dissertação. Terminados os trabalhos de arguição, o(a) Senhor(a) Presidente da Comissão Examinadora determinou a suspensão da sessão pelo tempo necessário ao julgamento da dissertação. Reunidos, em caráter secreto, no mesmo recinto, os membros da Comissão Examinadora passaram à apreciação da dissertação. Reaberta a sessão, o(a) Presidente da Comissão Examinadora anunciou o resultado do julgamento, tendo assim, o(a) candidato(a) obtido o Conceito APROVADO. Na sequência, o(a) Presidente da Comissão Examinadora anunciou o resultado do julgamento, tendo a seguir encerrado a sessão, da qual lavrei a presente ata, que vai assinada por mim, Lyana Silva e Cavalcante Nascimento, pelos membros da Comissão Examinadora e pelo(a) candidato(a). Campina Grande, 8 de Março de 2024.

Documento assinado eletronicamente por **LYANA SILVA E CAVALCANTE NASCIMENTO**, **ASSISTENTE EM ADMINISTRACAO**, em 12/03/2024, às 15:48, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **José Júnior Silva da Costa**, **Usuário Externo**, em 12/03/2024, às 15:50, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Márcio de Medeiros Ribeiro**, **Usuário Externo**, em 12/03/2024, às 20:35, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **4275440** e o código CRC **C95335B3**.

Referência: Processo nº 23096.015202/2024-15 SEI nº 4275440

Resumo

Ler e compreender o código são atividades cruciais durante o processo de evolução e manutenção de software. Porém, existem vários fatores que podem afetar esse entendimento, como a forma como o código está estruturado. Iteração, Recursão e Compreensão de Lista (LC) são algumas dessas técnicas de estruturação. Entretanto, seu impacto no desempenho dos desenvolvedores tem sido pouco investigado sob a perspectiva do esforço visual com rastreamento ocular, inclusive no contexto de novatos. Portanto, este trabalho tem como objetivo realizar um estudo com rastreamento ocular para investigar o impacto dessas diferentes estruturas na compreensão de código por novatos. Portanto, foi realizado um estudo inicial controlado para resolver seis tarefas, utilizando o desenho do Quadrado Latino com 32 novatos em Python, medindo o tempo, o número de tentativas para resolver a tarefa e o esforço visual através da duração da fixação, contagem de fixação e regressões. Foi utilizada uma comparação com tarefas que possuem as seguintes estruturas: Recursão, estrutura de repetição e LC. Na *Área de Interesse* (AOI), em relação às métricas analisadas, foram observados aumentos nas versões `while`, Recursão e LC em relação à versão `for`. No número de regressões, o aumento chegou a 100% e 114,29% com LC e `while`, respectivamente, indicando a necessidade de retornar mais vezes no código. O aumento no tempo chegou a 95% com `while`. Através da análise dos padrões de leitura, na Recursão, percebeu-se maior foco de atenção no caso base e na etapa recursiva. A necessidade de uma condição de parada e contador explícito é uma hipótese para o pior desempenho com `while`. Houve concordâncias e discrepâncias entre os participantes entre desempenho e percepção de dificuldade dependendo da tarefa. Em geral, a versão `for` exigiu menos tempo, tentativas e esforço visual, indicando melhor compreensão de algumas tarefas. Este estudo contribui principalmente para aumentar a conscientização entre educadores sobre o impacto da Recursão, Iteração e LC na compreensão do código para iniciantes em Python.

Abstract

Reading and understanding code are crucial activities during the software evolution and maintenance process. However, there are several factors that can affect this understanding, such as the way the code is structured. Iteration, Recursion and List Comprehension (LC) are some of these structuring techniques. However, its impact on developers' performance has been little investigated from the perspective of visual effort with eye tracking, including in the context of novices. Therefore, this work aims to conduct a study with eye tracking to investigate the impact of these different structures on novices' code understanding. Therefore, an initial controlled study was conducted to solve six tasks, using the Latin Square design with 32 Python novices, measuring time, number of attempts to solve the task and visual effort through of fixation duration, fixation count and regressions. A comparison was used with tasks that have the following structures: Recursion, repetition structure and LC. In the *Area of Interest* (AOI), regarding the metrics analyzed, increases were observed in the `while`, Recursion and LC versions compared to the `for` version. In the number of regressions, the increase reached 100% and 114.29% with LC and `while`, respectively, indicating the need to return more times in the code. The increase in time reached 95% with `while`. Through the analysis of reading patterns, in Recursion, greater focus of attention was noticed in the base case and in the recursive step. The need for a stop condition and explicit counter is a hypothesis for the worst performance with `while`. There were agreements and discrepancies among participants between performance and perception of difficulty depending on the task. In general, the `for` version required less time, attempts and visual effort, indicating a better understanding of some tasks. This study mainly contributes to raising awareness among educators about the impact of Recursion, iteration and LC on code understanding for Python novices.

Agradecimentos

Primeiramente, expresso minha gratidão a Deus por todas as bênçãos concedidas e por esta preciosa oportunidade.

À minha família, meu agradecimento pelo amor e pelo constante apoio ao longo desta jornada, oferecendo suporte e encorajamento em todos os momentos. Ao meu irmão José Aldo, minha gratidão pelas valiosas orientações, pela paciência e pelo apoio desde o início.

Ao meu orientador, Rohit Gheyi, sou grato por sua orientação excepcional, sua paciência e seu apoio ao longo deste percurso. Suas orientações me tornaram um profissional melhor.

Aos amigos e à comunidade da igreja, meu sincero obrigado por fazerem parte desta trajetória e por compartilharem comigo momentos memoráveis.

À Universidade Federal de Campina Grande e à Coordenação da Pós-graduação em Computação da UFCG (COPIN), expresso minha gratidão pelo apoio oferecido durante todo este percurso.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), agradeço pelo suporte financeiro fundamental que viabilizou a realização deste projeto acadêmico.

Meu obrigado.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code comprehension can be understood as the ability to understand and interpret the source code of a computer program. This involves the ability to read, analyze and understand what the code is doing and how it is structured. According to Xia et al. [73], code understanding, also known as program comprehension, refers to the active process in which developers acquire knowledge about a software system through investigating and exploring software artifacts. This process involves reading the relevant code and/or documentation. It is crucial for improving code, for identifying and fixing errors, for software maintenance and updating, for team collaboration, code reuse, and performance.

Furthermore, reading code is an activity performed more frequently than writing [71]. Developers often spend more time reading code than writing it, which makes readability and understanding crucial. For example, a recent large-scale study concluded that understanding code takes an average of 58% of developers' time and that the percentage of time spent by novice programmers is greater than that of more experienced programmers [73].

1.1 Problem

Therefore, the different ways of structuring code can impact code understanding, especially for novice programmers. In this way, an organized code that is easy to understand and maintain can contribute to a positive impact on the efficiency and consequently on the expenses of software projects.

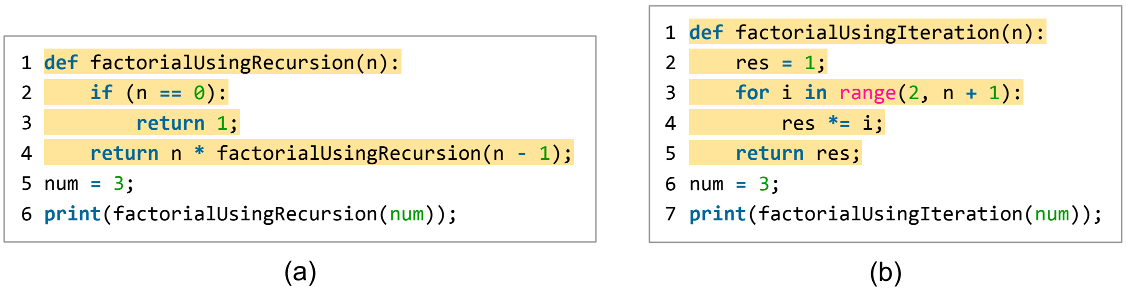
However, there is a need to better understand how developers read and understand code.

There are still few studies comparing these different structures and their impact on code understanding. Some example studies investigate code understanding with MRI [62], and [70] and [49]. Furthermore, the few that exist do not employ a method that allows us to understand this impact in detail from the point of view of visual effort.

1.2 Motivating Example

In software development, several factors can influence code understanding. Clarity of implementation, developer experience and familiarity with the concepts, among other good programming practices, are examples of these. The choice of code structures can influence the understanding of the code and generate discussions about which one to use. Iteration and Recursion, although both approaches have advantages and disadvantages, have generated discussions about which one to use depending on the problem in question.

Despite the contribution of the works already carried out, there is still much to be investigated on this topic, especially with regard to the use of more quantitative metrics, which allows us a more precise and in-depth analysis in terms of understanding. Furthermore, there is a need to investigate more dynamic aspects that go beyond time and accuracy or code metrics, such as the use of visual metrics to understand the programmer's effort.



```python
def factorialUsingRecursion(n):
    if (n == 0):
        return 1;
    return n * factorialUsingRecursion(n - 1);
num = 3;
print(factorialUsingRecursion(num));
```

(a)

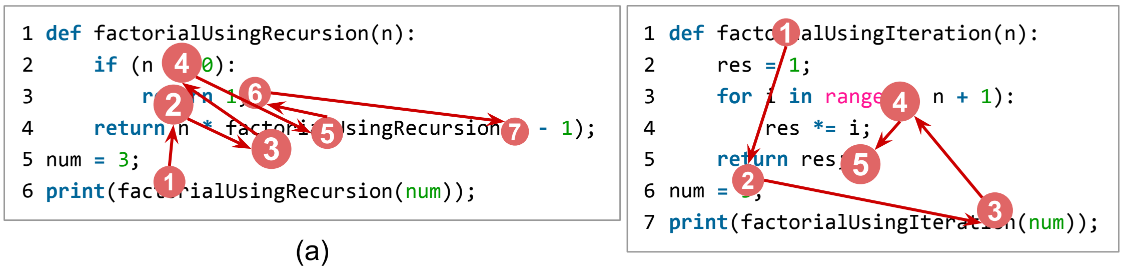
```python
def factorialUsingIteration(n):
    res = 1;
    for i in range(2, n + 1):
        res *= i;
    return res;
num = 3;
print(factorialUsingIteration(num));
```

(b)

Figure 1.1: Comparison of the code to calculate the factorial of a number using the structures of (a) Recursion and (b) Iteration, adapted from *GeeksforGeeks*.

In Figure 1.1, two code examples are presented to solve the factorial of a number. The code were found and adapted from the website *Geeksforgeeks*.[1] On the left side, Figure 1.1(a), the version with iteration is presented, and on the right side Figure 1.1(b), the

[1]www.geeksforgeeks.org



```
1 def factorialUsingRecursion(n):
2     if (n == 0):
3         return 1;
4     return n * factorialUsingRecursion(n - 1);
5 num = 3;
6 print(factorialUsingRecursion(num));
```

```
1 def factorialUsingIteration(n):
2     res = 1;
3     for i in range(1, n + 1):
4         res *= i;
5     return res;
6 num = 3;
7 print(factorialUsingIteration(num));
```

Figure 1.2: Representation of fixations and transitions in code comparison to calculate the factorial of a number using the structures of (a) Recursion and (b) Iteration.

version with Recursion. Both codes display the same result when executed.

In Figure 1.2, an example of a sequence of fixations is presented, in which the red circles vary in size according to the fixation time. In Figure 1.2(a) the version with Recursion is presented, in which the participant fixated on seven different locations in the code, six of which were in the area of interest, lines 1–4. In Figure 1.2(b) the version with Iteration is presented, in which the participant fixated on five different locations, three of which were in the area of interest, lines 1–5. In the Recursion version, there were more fixations and regressions, along with long-term fixations.

Analyzing different repetition structures through the visual behavior of novices provides important insights into the most critical and necessary points of attention for novices to understand code. Eye tracking makes it possible to analyze visual effort, making it possible to detect new insights that would not be possible with common metrics.

1.3 Solution

To address the gap of exploring more dynamic aspects that go beyond time and accuracy or code metrics to understand the programmer's effort, our study aims to explore the impact of different code structures – specifically, Iteration, Recursion, and List Comprehension (LC) – on the comprehension abilities of novice Python programmers. By leveraging eye-tracking technology, we intend to provide a detailed analysis of the visual effort exerted by novices when interacting with these structures. This approach is expected to yield valuable insights into the cognitive processing of code, thereby informing both pedagogical strategies and software development practices.

1.4 Evaluation

Our research methodology involves conducting a controlled study with 32 Python novices using an eye-tracking camera to capture their movements. We intend to assign six program tasks to the novices using the Latin Square design for task distribution. The study's design allows us to isolate and evaluate the effects of different code structures on comprehension. We focus on measuring several key metrics, including time taken to solve tasks, number of attempts, and varied indicators of visual effort such as fixation duration, fixations count, and regressions count. These metrics provide a comprehensive view of the cognitive load and visual effort associated with each code structure.

1.5 Conclusions

Our initial findings indicate significant differences in visual effort and comprehension efficiency among the different code structures. Notably, tasks involving `while` loops, Recursion, and LC showed an increase in visual effort compared to those using the `for` loop. These results suggest a more complex cognitive process for certain structures, as evidenced by increased times and higher frequencies of code revisiting. This research contributes significantly to the field of software engineering education, particularly in understanding the pedagogical implications of teaching various code structuring techniques. By highlighting the specific challenges associated with Iteration, Recursion, and LC, especially for novices, this study provides educators and developers with critical insights into optimizing teaching methods and improving code comprehension among novice programmers.

1.6 Summary of contributions

The main contributions of this work consists of:

- A controlled experiment with 32 novices in Python aiming to evaluate the impact of different code structures (Iteration, Recursion, and LC) on the comprehension abilities of novice Python programmers (Chapter 3);

- We present and discuss the eye tracking method to evaluate Iteration, Recursion and LC, especially for novices (Chapter 4).

1.7 Organization

This work is organized as follows: In Chapter 2, we provide a background. In Chapter 3, we describe the methodology. In Chapter 4, we describe the evaluation of our technique, results and discussion. In Chapter 5, we present the related works. In Chapter 6, we present the conclusion of this work.

Chapter 2

Background

In this chapter, we provide a comprehensive background to enhance the reader's understanding of the fundamental concepts employed in our study. This chapter is divided into two main sections focusing on Repetition Structures in Python (Section 2.1), Eye tracking Technology (Section 2.2) and Code comprehension with eye tracking (Section 2.3).

2.1 Repetition Structures in Python

In this section, we present the four types of repetition structures in Python mentioned in this work: `while` structure (Section 2.1.1), `for` structure (Section 2.1.2), Recursion structure (Section 2.1.3), and List Comprehension structure (Section 2.1.4).

2.1.1 **`while`** structure

The `while` structure applies to situations where the number of repetitions is not predetermined. Its dynamic nature allows it to continue executing as long as the specified condition holds *True*, offering flexibility in adapting to changing circumstances during runtime.

The `while` structure in Python is similar to other programming languages such as in C and Java, where a condition is tested, and if true, the instructions in the following set are executed in a loop. These instructions potentially change the state of the condition, so the condition is tested again and the set is potentially executed again. This process continues until a certain condition occurs, and if the condition is initially `False`, the set will never be executed. If the condition does not become `False`, the loop continues and never ends [2].

In Figure 2.1, we present an example of repetition structure `while`, which was found and adapted from the website *Geeksforgeeks*. On the left side, we can analyze the syntax, and on the right side, a simple code example. In this example, the loop is performed three times, while incrementing the *counter* variable. In this example, we can identify the main characteristics of `while` loop structure.

```
while expression:

   statement(s)
```

(A)

```
count = 0
while (count < 3):
    count = count + 1
```

(B)

Figure 2.1: Example of `while` repetition structure, found and adapted from the website *Geeksforgeeks*. In it, (a) represents the syntax while (b) represents a code snippet.

One potential problem using `while` loop is the risk of creating infinite repetitions. If the loop condition is not carefully managed, it may never be evaluated to *False*, leading to continuous execution. Thus, the programmer must be cautious when initializing and updating the loop variables to prevent this issue. This may also pose challenges to the understanding of the structure since the programmer must be aware of the state of the variable.

However, in scenarios where the number of repetitions is predetermined, such as iterating over a fixed list of elements, alternative loop structures, such as `for` loop might offer an alternative solution. Choosing the appropriate loop structure depends on the specific requirements of the task at hand.

2.1.2 `for` structure

The `for` structure plays a crucial role in iterating over iterable structures. This loop structure is specifically designed for scenarios where the number of repetitions is predetermined, making it particularly useful when iterating over fixed lists of elements such as lists, tuples, dictionaries, or strings. The `for` loop uses values derived from iterable structures. The iterable serves as a source for a sequence of values, each linked to the loop variable. With each iteration, the set of instructions within the loop is executed after assigning the current value to the loop variable [2].

In Figure 2.2 an example of a repetition structure `for` is presented, found and adapted from the website *Geeksforgeeks*. On the left side we can analyze the syntax and on the right side a simple code example. In this example, the loop is executed four times, while performing a print on each execution. In this example we can identify the main characteristics of `for`.

```
for iterator_var in sequence:
    statements(s)
```

(A)

```
n = 4
for i in range(0, n):
    print(“”)
```

(B)

Figure 2.2: Example of repetition structure `for`, found and adapted from the website *Geeksforgeeks*.

Similar to the `while` loop, the `for` loop presents its potential challenges that require being cautious during implementation. One key consideration is to avoid modifying the sequence being iterated over within the loop. Modifying the underlying sequence can result in unexpected and unintended consequences, potentially leading undesired behavior in the loop.

For instance, when we iterate over a list of elements using a `for` loop and modify that list by adding or removing elements within the loop, it can disrupt the intended flow of iterations. The loop may not behave as expected since the sequence it is iterating over is dynamically changing, which can have an impact on developers' code understanding.

Each loop structure has its advantages and is better suited to specific scenarios. The choice between `while` and `for` depends on the nature and the characteristics of the scenario. While structures such as `while` and `for` can have advantages, in some cases, recursion structure can reduce redundant code, especially when dealing with repetitive patterns and complex data structures.

2.1.3 Recursion structure

Iteration involves controlled repetition, so a set of instructions need to be executed one by one until a condition is satisfied. This is an approach that is considered easy to understand and implement. On the other hand, Recursion involves calling a function by itself repeatedly, in which a set of instructions are executed until a certain condition is satisfied. This is an approach considered more challenging to understand, requiring more abstract logical reasoning, but in some contexts it can provide a clearer solution to complex problems.

In Figure 2.3 an example of a Recursion repetition structure is presented, found and adapted from the website *Geeksforgeeks*. On the left side we can analyze the syntax and on the right side a simple code example. In this example, recursion is used to calculate a factorial of a value *n*, where the function calls itself at each execution, through the recursive step, and each calculation performed waits for the result of the next calculation. When the base case is satisfied, the value that satisfies each calculation is returned. In this example we can identify the main characteristics of Recursion.

```
def func(): <--
               |
               | (recursive call)
               |
      func() ----
```

(A)

```
def recursive_factorial(n):
  if n == 1:
      return n
  else:
      return n *
recursive_factorial(n-1)
```

(B)

Figure 2.3: Example of repetition structure Recursion, found and adapted from the website *Geeksforgeeks*.

2.1.4 List Comprehension structure

One of the new features of the Python language is the possibility of generating *tuple*, *list*, *dictionary* and *set* from loops in structures that are iterable. These runtime constructs are called *comprehensions* [2].

In Figure 2.4 an example of an LC repetition structure is presented, found and adapted from the website *Geeksforgeeks*. We can look at a simple code example. In this example, LC is used to calculate the square of each value in the list *numbers*, while creating and saving the new values, and at the end printing the list with the new values. Code with LC tends to be summarized and present the main points of the code in one line. In this example we can identify the main characteristics of LC.

```
numbers = [1, 2, 3]
squared = [x ** 2 for x in numbers]
print(squared)
```

Figure 2.4: Example of repetition structure LC, found and adapted from the website *Geeksforgeeks*.

2.2 Eye tracking Technology

This section is divided into Eye tracking methodology (Section 2.2.1), metrics (Section 2.2.2) and visualization techniques (Section 2.2.3).

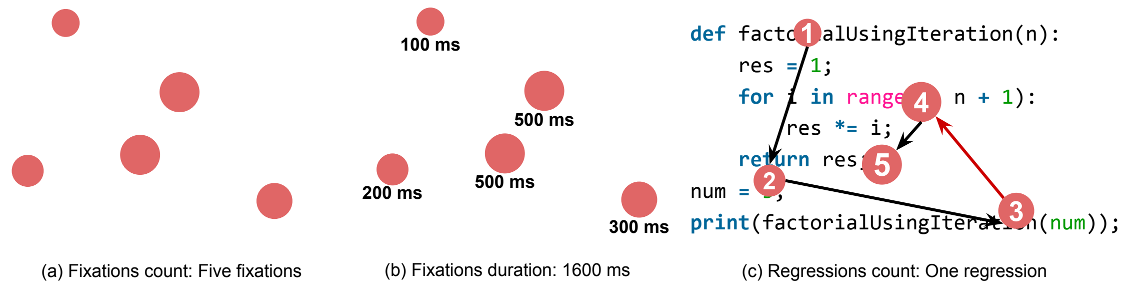
2.2.1 Eye tracking methodology

Eye trackers help in the analysis of visual attention, recording eye movements, being able to identify where the participant is looking, the duration and sequence of changes in attention locations [57]. In Figure 1.1 two codes were presented, which perform the same calculation, the factorial of a number and when executed they display the same result. The codes were found and adapted from the website *Geeksforgeeks*. On the left side, Figure 1.1(a), we can observe the version written with iteration, while on the right side Figure 1.1(b), we can observe the version with Recursion.

For the analysis of code comprehension, in Figure 1.1, in addition to commonly used metrics such as time and responses, it can be analyzed from other perspectives, such as visual effort. Identifying the places where the most time and visual effort are spent in the code, the most difficult places and most critical parts are examples of the possibilities using eye tracking.

To analyze visual effort, the following metrics can be used: number of fixations, which corresponds to the number of times the participant fixed their gaze on the screen for at least 200 milliseconds; the duration of fixations, which consists of the sum of the time spent at all times in which the participant made a fixation; and the number of regressions, which corresponds to the number of times the participant returns within the code snippet.

Some studies have been carried out with the aim of better understanding the impact that code structures have on code comprehension [21; 68; 32]. However, these works use subjective metrics such as interviewees' opinions, preferences and classroom evaluations. Some works use more quantitative metrics such as accuracy and time [16; 19].


100 ms
500 ms
500 ms
200 ms
300 ms
1
2
3
4
5
(a) Fixations count: Five fixations
(b) Fixations duration: 1600 ms
(c) Regressions count: One regression

```
def factorialUsingIteration(n):
    res = 1;
    for i in range[illegible] n + 1):
        res *= i;
    return res;
num = [illegible];
print(factorialUsingIteration(num));
```

Figure 2.5: Example of fixations and regressions metrics

2.2.2 Eye tracking Metrics

Eye tracking metrics can be divided into four categories: metrics based on fixation, metrics based on saccade, metrics based on scanpath, and metrics based on pupil size and blink rate [57]. The metrics based on fixation used in this work are briefly explained.

- **Duration of fixations** - While examining a scene, our eyes maintain stability for a certain duration, enabling us to concentrate on specific elements. As soon as we see a word or a piece of text, for instance, we try to interpret it, directing our attention toward it until we understand it [35]. The position and duration of fixations have been associated with the focus of attention [14]. For instance, longer fixations have been associated with an increase in demands of attentiveness [12]. In Figure 2.5(b), an example of the duration of fixations is presented.

- **Number of fixations** - It refers to our ability to fixate our eyes on different locations to examine a scene. An increased number of fixations is indicative of an extended processing time required to comprehend code as well as more attention to the code [7; 13]. A high number of fixations is associated with a greater visual effort to answer a question [59]. In Figure 2.5(a), an example of the fixation count is shown.

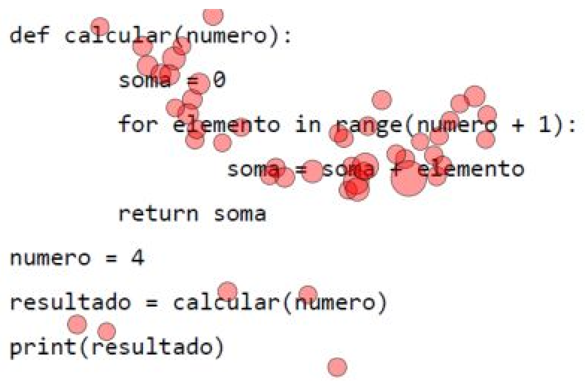
- **Eye movement Regressions** - These regressions consist of backward eye movements over the stimuli [11]. Thus, in the code, it can be understood and visually returning in the code. Regressions can be used as a metric for visual effort [57]. In Figure 2.5(c), an example of regression counting is presented.

2.2.3 Visualization techniques

The analysis of data generated from eye trackers allows for quantitative and qualitative visual analysis. More recently, there has been an effort to propose methods for visualization of the gaze patterns and behaviors of software developers [18]. Despite the variety of data visualization techniques, studies commonly employ three main types: gaze plots, heatmaps, and gaze transitions to obtain important insights.

Gaze plots provide a static view of the eye-gaze data and show the time sequence of looking using the locations, orders, and duration of fixations on stimuli [58]. Figure 2.6 presents an example of a repetition structure `for` with a gaze plot. In this example, `for` is used in a function to calculate the factorial of a number, and at the end prints the result returned by the function. We can identify the places where the participant focused on the code through the analysis of the fixations. For instance, a fixation is represented by a red dot. The larger it gets, the more time the subjects spend focusing on that location.

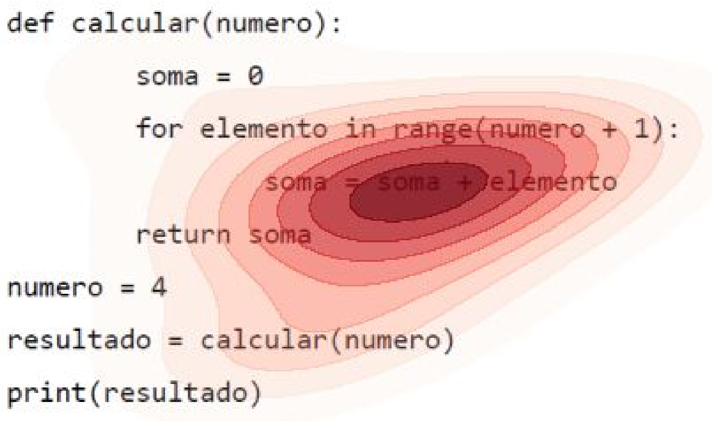
Heatmaps consist of two-dimensional graphical representations of data that depict variable values using colors. These representations are useful to assess the level of interest elicited by different elements of the stimulus. For instance, in the code scenario, they can represent the distribution of visual attention [9]. Through colors, this visualization represents the intensity of a measure, such as the number of fixations received by a stimulus [58]. In Figure 2.7, we present an example of a repetition structure `for` with the heatmap. In this



```
def calcular(numero):
        soma = 0
        for elemento in range(numero + 1):
                soma = soma + elemento
        return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 2.6: Example of gaze plot with red circles representing fixations that vary in size according to their duration.

example, `for` is used to calculate the factorial of a number, and at the end prints the result. In this example, we can identify the places where fixations are concentrated, that is, where the participant focused on the code most of the time, and the redder the more time was spent focusing on that place in the code.

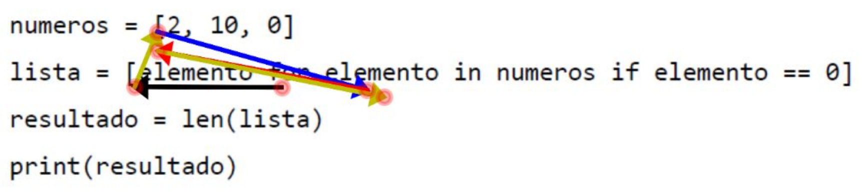


```
def calcular(numero):
        soma = 0
        for elemento in range(numero + 1):
                soma = soma + elemento
        return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 2.7: Example of heat map.

Gaze transitions consist of eye movement transitions from one fixation to another, also called *saccades* [55; 51]. Besides counting the number of transitions, researchers have tracked the chronological order of these transitions to infer a path made by the eyes [58]. In

Figure 2.8, we present an example of LC repetition structure with gaze transitions. In this example, LC is used to find out how many zero values there are in a list and finally print the result. In this example, we can identify the path that the participant visually took in the code. We can identify that the participant goes back and forth several times between the list and the line that is `for`.



```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

Figure 2.8: Example of gaze transitions with arrows indicating the direction of the transition.

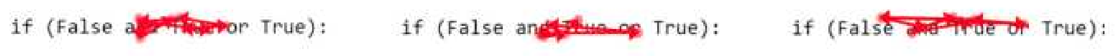
2.3 Code comprehension with eye tracking

Crosby and Stelovsky [15], one of the pioneering works in eye movement analysis for code comprehension, explored the way in which subjects saw an algorithm, written in Pascal, and the graphical representation of visual behavior.

Several studies have applied eye tracking in code comprehension [33; 36; 61], which has been a promising area of research. Researchers have provided valuable insights into how developers visually interact with the code and how much effort certain code structures require from the developers. Controlled experiments have explored how eye tracking can be used to analyze reading patterns, identify areas of focus, and understand the nuances of the code comprehension process.

2.3.1 Analysis of Reading Patterns

A controlled experiment with eye trackers has captured developers' eye movements while they examine the code. Such data have allowed the analysis of fixation and saccade patterns. For instance, Da Costa et al. [16; 63] investigated atoms of confusion, small snippets that confuse developers [40]. In their study, they searched for gaze patterns in the paths followed during reading that indicate confusion, such as depicted in Figure 2.9, found in the study of Da Costa et al. [16]. They observed transitions going forward and backward between `'True or True'` which may indicate that the subject has doubts about which expression should be evaluated first without the parentheses.


if (False and True or True):
if (False and True or True):
if (False and True or True):

(a) Transitions of Subject 2 (b) Transitions of Subject 2 (c) Transitions of Subject 10

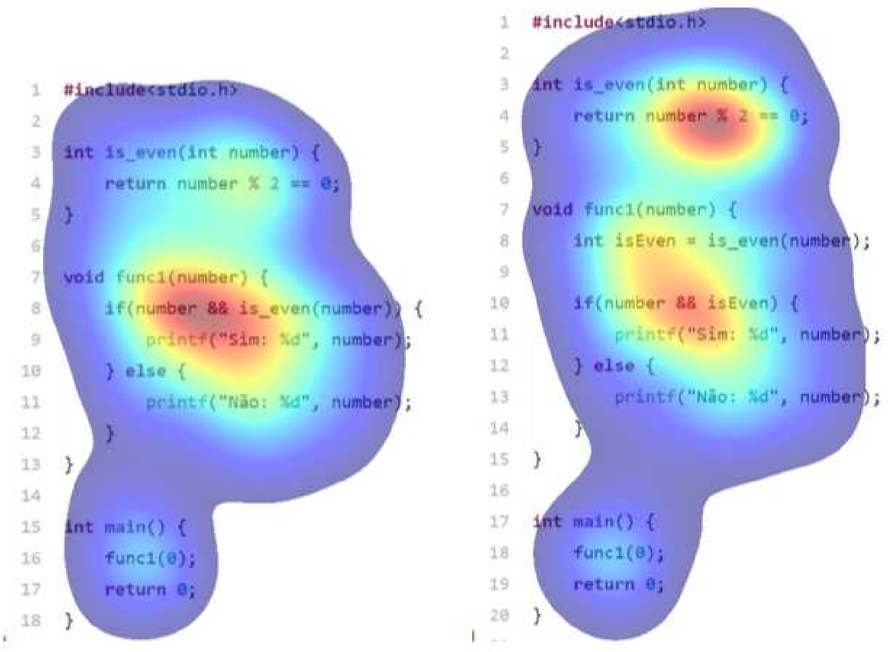
Figure 2.9: Example of gaze transitions of developers on the code with an atom of confusion.

Other works have also investigated these reading patterns, for example, Blascheck and Sharif [8] investigate how people read text in natural language compared to source code.

2.3.2 Focus of Attention

Eye movements focus the subject's visual attention on the parts of a visual stimulus that are processed by the brain, triggering the cognitive processes that are required to perform tasks [35]. Understanding where the eyes concentrate during code analysis reveals crucial information about the focus of attention. For example, identifying whether developers focus more on flow control statements, variable declarations, or specific code sections can inform reading strategies.

In another controlled experiment, Oliveira et al. [47] studied the focus of attention of developers while they examined code in C in the presence of atoms of confusion. According to the authors, the concentration of attention was primarily on a single focal area where the atom is situated (left side of Figure 2.10, found in the study of Oliveira et al. [47]). However, when they removed the atom, developers shifted their attention towards two main regions, leading to a more dispersed distribution of attention across distinct parts (right side of Figure 2.10, found in the study of Oliveira et al. [47]). Other eye tracking studies have investigated attention on code comprehension activities as well, such as how the attention allocation and its switching between code areas [4; 12]



```
#include<stdio.h>

int is_even(int number) {
    return number % 2 == 0;
}

void func1(number) {
    if(number && is_even(number)) {
        printf("Sim: %d", number);
    } else {
        printf("Não: %d", number);
    }
}

int main() {
    func1(0);
    return 0;
}
```

```
#include<stdio.h>

int is_even(int number) {
    return number % 2 == 0;
}

void func1(number) {
    int isEven = is_even(number);

    if(number && isEven) {
        printf("Sim: %d", number);
    } else {
        printf("Não: %d", number);
    }
}

int main() {
    func1(0);
    return 0;
}
```

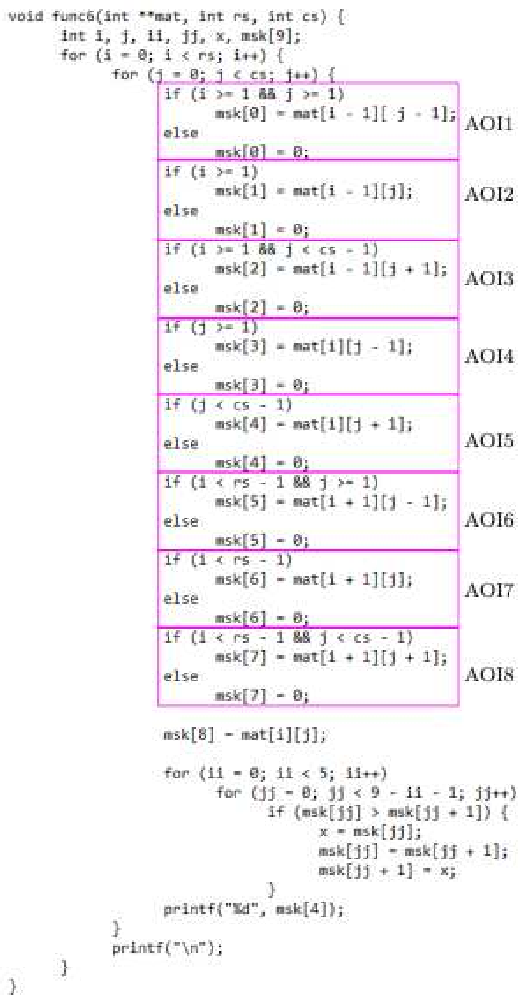
Figure 2.10: Example of focus of attention of developers on the code with an atom of confusion (left hand side) and without the atom (right hand side).

2.3.3 Identifying Comprehension Difficulties

Eye tracking can be instrumental in identifying areas of code that pose comprehension difficulties. If certain code lines result in inconsistent reading patterns, this may indicate critical points that warrant closer examination.

Studies have already been carried out using eye tracking to carry out investigations in this field. Bednarik and Tukiainen were pioneers in utilizing eye tracking as a tool for analyzing cognitive processes during program comprehension [4]. Peitek et al. [50] explored the feasibility of incorporating simultaneous eye tracking alongside fMRI measurements aiming to enhance the explanatory power of fMRI measurements for programmers. Sorg et al. [66] investigated critical parts of the code that may be associated with cognitive load. Using ocular fixation resources, they qualitatively investigate the relationship with parts perceived as challenging by users.

Jbara and Feitelson [34] investigated whether regularity in the code makes it more difficult to understand to programmers. Regularity consists of the repetition of code patterns such as a certain pattern of nested control statements, where repeated instances of the pattern are usually successive, such as in Figure 2.11 found in the study of Jbara and Feitelson [34]. They found that the programmers tend to invest more effort on the initial repetitions, and less and less on successive ones.



```
void func6(int **mat, int rs, int cs) {
     int i, j, ii, jj, x, msk[9];
     for (i = 0; i < rs; i++) {
          for (j = 0; j < cs; j++) {
               if (i >= 1 && j >= 1)
                    msk[0] = mat[i - 1][ j - 1];
               else
                    msk[0] = 0;
               if (i >= 1)
                    msk[1] = mat[i - 1][j];
               else
                    msk[1] = 0;
               if (i >= 1 && j < cs - 1)
                    msk[2] = mat[i - 1][j + 1];
               else
                    msk[2] = 0;
               if (j >= 1)
                    msk[3] = mat[i][j - 1];
               else
                    msk[3] = 0;
               if (j < cs - 1)
                    msk[4] = mat[i][j + 1];
               else
                    msk[4] = 0;
               if (i < rs - 1 && j >= 1)
                    msk[5] = mat[i + 1][j - 1];
               else
                    msk[5] = 0;
               if (i < rs - 1)
                    msk[6] = mat[i + 1][j];
               else
                    msk[6] = 0;
               if (i < rs - 1 && j < cs - 1)
                    msk[7] = mat[i + 1][j + 1];
               else
                    msk[7] = 0;

               msk[8] = mat[i][j];

               for (ii = 0; ii < 5; ii++)
                    for (jj = 0; jj < 9 - ii - 1; jj++)
                         if (msk[jj] > msk[jj + 1]) {
                              x = msk[jj];
                              msk[jj] = msk[jj + 1];
                              msk[jj + 1] = x;
                         }
               printf("%d", msk[4]);
          }
          printf("\n");
     }
}
```

Figure 2.11: Regular code implementation with AOIs investigated.

2.3.4 Evaluating the Impact of Code Layout and Structure

Examining how code layout and structure affect eye movements provides insights into the influence of layout on comprehension. This may include assessing how indentation, spacing, and code block structuring impact reading efficiency.

Saddler et al. [54], using eye tracking, investigated reading behavior in terms of how posts on the Stack Overflow forum are structured and discussed how observations can benefit the way users structure their posts. They also investigated which elements developers read on pages and how specific attributes of posts, that is, code block count and paragraph count, impact gaze behavior. Abid et al. [1] investigated, using eye tracking, the mental models applied during program comprehension, specifically investigating bottom-up and top-down.

Studies using eye tracking have also compared the impact of layout on the understanding of experts and novices. Sharafi et al. [58], investigated the effect of layout on understanding the roles of design patterns in UML class diagrams, as shown in the Figure 2.12 found in the study of Sharafi et al. [58].

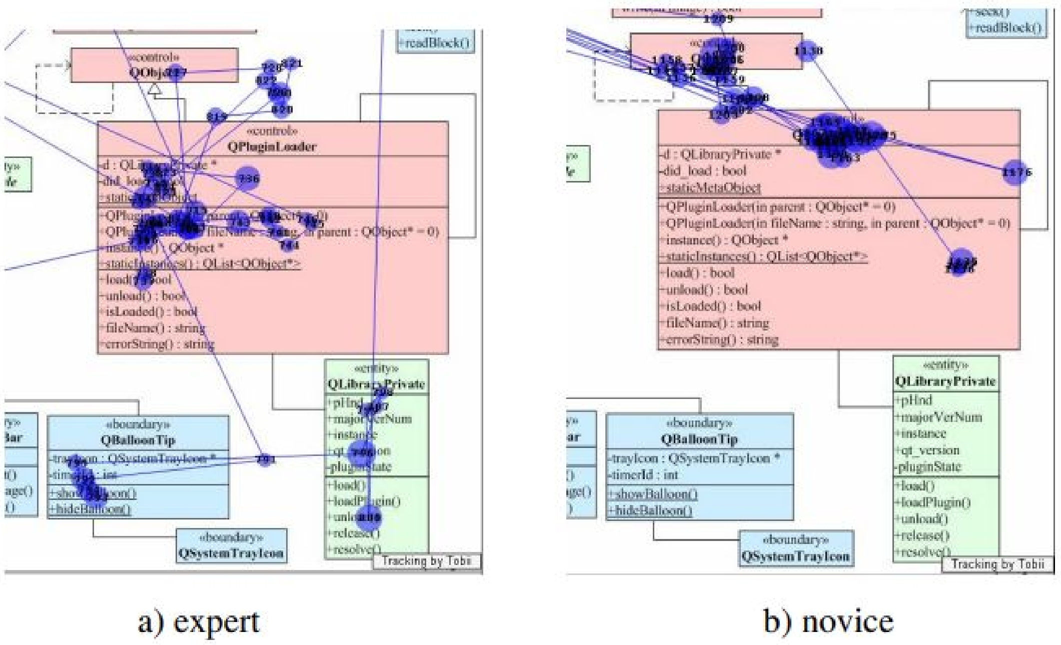
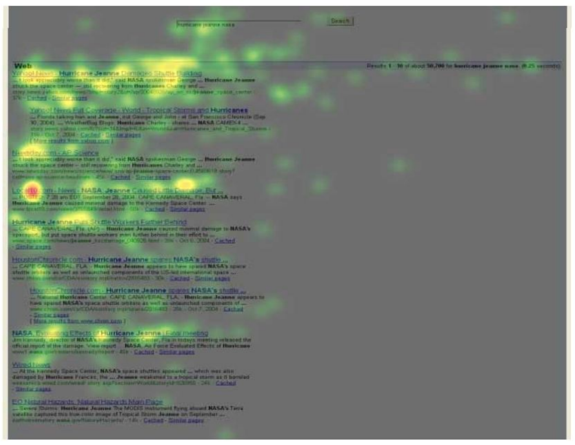


Figure 2.12: Gaze plot comparison of an expert and a novice.

2.3.5 Adapting Development Interfaces and Tools

Based on eye tracking findings, it is possible to adapt development interfaces and tools to better meet the needs of programmers. This may involve layout adjustments, design suggestions, or the implementation of specific features to enhance the reading experience.

Rele and Duchowski [52] investigated two types of search results interfaces, measuring performance and studying eye behavior using an eye tracker. The two interfaces used were the list interface and a tabular interface (Figure 2.13 found in the study of Rele and Duchowski [52]). The results showed no significant differences in performance between the interfaces, but eye movement analysis provides some insights into the title, summary and URL. Other works has already investigated the use of eye tracking to assist developers, which can be used in IDEs and even make code recommendations based on the developer's eye movements [72; 30; 56]. In Figure 2.14 found in the study of Walters et al. [72], we can see a screenshot of the Itrace layout, where artifacts are shown side by side so that the user can look between them.


(A) List interface

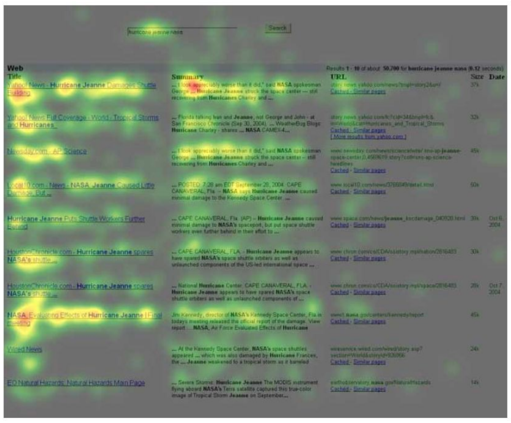
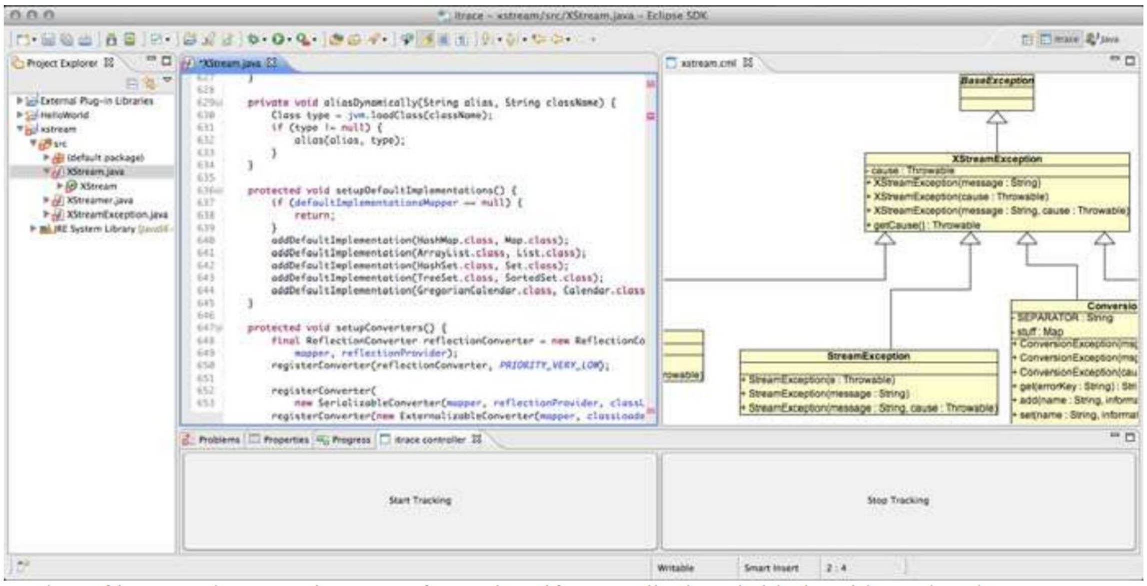

(B) Tabular interface

Figure 2.13: Heatmap comparison of two interfaces, list interface and a tabular interface.

2.3.6 Integration with Performance Metrics

By combining eye tracking data with traditional programming performance metrics, such as task completion time and accuracy, a more comprehensive understanding of the code comprehension process can be obtained.

Fritz et al. [26] for instance, investigated the use psychophysiological measurements to determine whether a code comprehension task is perceived as easy or difficult. They com-



```
    }

    private void aliasDynamically(String alias, String className) {
        Class type = jvm.loadClass(className);
        if (type != null) {
            alias(alias, type);
        }
    }

    protected void setupDefaultImplementations() {
        if (defaultImplementationsMapper == null) {
            return;
        }
        addDefaultImplementation(HashMap.class, Map.class);
        addDefaultImplementation(ArrayList.class, List.class);
        addDefaultImplementation(HashSet.class, Set.class);
        addDefaultImplementation(TreeSet.class, SortedSet.class);
        addDefaultImplementation(GregorianCalendar.class, Calendar.class
    }

    protected void setupConverters() {
        final ReflectionConverter reflectionConverter = new ReflectionCo
            mapper, reflectionProvider);
        registerConverter(reflectionConverter, PRIORITY_VERY_LOW);

        registerConverter(
            new SerializableConverter(mapper, reflectionProvider, classL
        registerConverter(new ExternalizableConverter(mapper, classLoade
```

```
BaseException

XStreamException
- cause : Throwable
+ XStreamException(message : String)
+ XStreamException(cause : Throwable)
+ XStreamException(message : String, cause : Throwable)
+ getCause() : Throwable

StreamException
+ StreamException(e : Throwable)
+ StreamException(message : String)
+ StreamException(message : String, cause : Throwable)

Conversio
- SEPARATOR : String
- stuff : Map
+ ConversionException(ms
+ ConversionException(ms
+ ConversionException(cau
+ get(errorKey : String) : Str
+ add(name : String, informa
+ set(name : String, informat
```

Figure 2.14: A screenshot of the Itrace layout. Display of artifacts side by side so that the user can look between them.

bined eye tracking with electroencephalographic and electrodermal activities.

However, one has to be careful when designing controlled experiments that involve human subjects and use performance metrics to measure code comprehension phenomenon, since different factors can influence it. Feitellson [24; 25] discusses several factors concerning the experimental subjects, the source code they work on, the tasks they are asked to perform, and the metrics for their performance.

Chapter 3

Methodology

In this chapter, we present the methodology used to develop the work and achieve the objectives described in Section 4.1. This chapter is divided into pilot study (Section 3.1), experiment phases (Section 3.2), subjects (Section 3.3), Treatments (Section 3.4), programs (Section 3.5), and eye tracking system (Section 3.6).

3.1 Pilot Study

The participants were all native Brazilians. We used the vocabulary of the programs in Brazilian Portuguese, thus avoiding obstacles in understanding the vocabulary of the programs. The names of the methods and variables were selected and discussed by the researchers. Names such as "result" were used to receive the results of operations. We sought to avoid names that made it too easy to carry out the operations, opting for more neutral names such as "calculate", with the aim of having the participant analyze the code.

We adjusted and refined the names of the methods and variables in the pilot studies, testing how well the names presented the intention of the methods. The names were discussed by the researchers to find the best options. The experiment was organized into five phases: (1) Characterization, (2) Tutorial, (3) Warm-up, (4) Tasks and (5) Qualitative interview. The experiment was estimated to take around 60 minutes for each participant to complete all phases. The phases will be detailed below.

3.2 Experiment phases

In the first phase, a brief explanation is made about the study, how and what data will be captured. At this stage, each participant fills out a consent form agreeing to participate and is aware that their identity will remain anonymous. They also fill out a characterization form with questions about their experience with programming.

In the second phase, a tutorial is made explaining how to carry out the experiment. In this phase, participants are presented with instructions about the eye tracking camera and how the tasks should be carried out. After this, the camera is calibrated in the participant's eyes. For calibration, the participant must look at the locations on the screen indicated by the camera software. At the end of the calibration, the camera software also reports when the calibration was successful.

In the third phase, each participant warms up for the experiment by solving a simple problem. During the warm-up, it is demonstrated how to respond to the code output aloud. In addition, participants are also instructed to close their eyes for two seconds before and after solving the problem, and how it will be signaled whether the answer is correct or incorrect. After warming up, participants can become more comfortable with the setup of the experiment and the equipment used.

In the fourth phase, the experiment is carried out with six programs. To avoid the learning effect, the Latin Square design [10] is used, which will be explained in more detail in Section 3.4. In the fifth phase, the experiment ends with a semi-structured interview. This interview investigates how the participant examines the programs and what their impressions were. For each program three questions are asked:

- How did you find the program output? What strategy did you use?
- How do you evaluate the difficulty of the task: very easy, easy, neutral, difficult or very difficult?
- What were the main difficulties involved, if any? Could you point them out in the program?

To carry out the experiment, a fixed chair was used, which favors the accuracy of the eye tracking equipment. However, given camera limitations, perfect data capture is impossible. For mitigation, the data was plotted, discussed and a data correction was performed by moving fixation blocks on the y-axis. This strategy will be discussed in Section 4.4.

3.3 Subjects

Initially, for the pilot study, six undergraduates were recruited. After some adjustments to the experiment, four more were recruited, totaling six tests in the pilot study with 10 participants. They reported having between 6 and 48 months of experience with programming languages in general, including mainly Python, Java and C. Participants were recruited from two different universities in one city in Brazil, invited mainly in person. Participants were Portuguese speakers, enrolled in universities. The participants were mostly from the initial periods, but there was a diversification, with three from the second semester, one from the third, one from the fourth, one from the fifth, one from the seventh, one from the eighth, one from the eleventh and one from the twelfth.

For the experiment, 34 novices were recruited, however, 32 participants were evaluated. We consider undergraduate students to be novices. They reported having between 6 and 120 months of experience with programming languages in general. In some cases, participants had more experience because they had already been programming before starting their undergraduate course. Participants were recruited from three different universities in two cities in Brazil, invited mainly in person. Participants were Portuguese speakers, enrolled in universities. Participants recruited were from different semesters.

3.4 Treatments

As illustrated in Figure 3.1, each participant analyzed six programs (P1-P6). To avoid a learning effect, the Latin Square [10] design was used. Twelve different programs were designed, which were divided into two sets of programs (SP1 and SP2). A participant analyzes three programs from the SP1 set, namely `for`, `for` and `for`, and three programs from the SP2 set, namely `while`, Recursion and LC. Another participant analyzes three programs from the SP1 set, namely `for`, `for` and `for`, and three programs from the SP2 set, namely `while`, Recursion and LC. Programs that are in the same set although have different codes result in the same output. The programs with `for`, `for` and `for`, were designed to be the baseline group (B), and the `while`, Recursion and LC programs to be the treatment group (T). In all programs, participants must specify the correct output, but without multiple answer options. Given the program's input, the participant needs to perform tasks such as calculating factorial, list sum, among others.

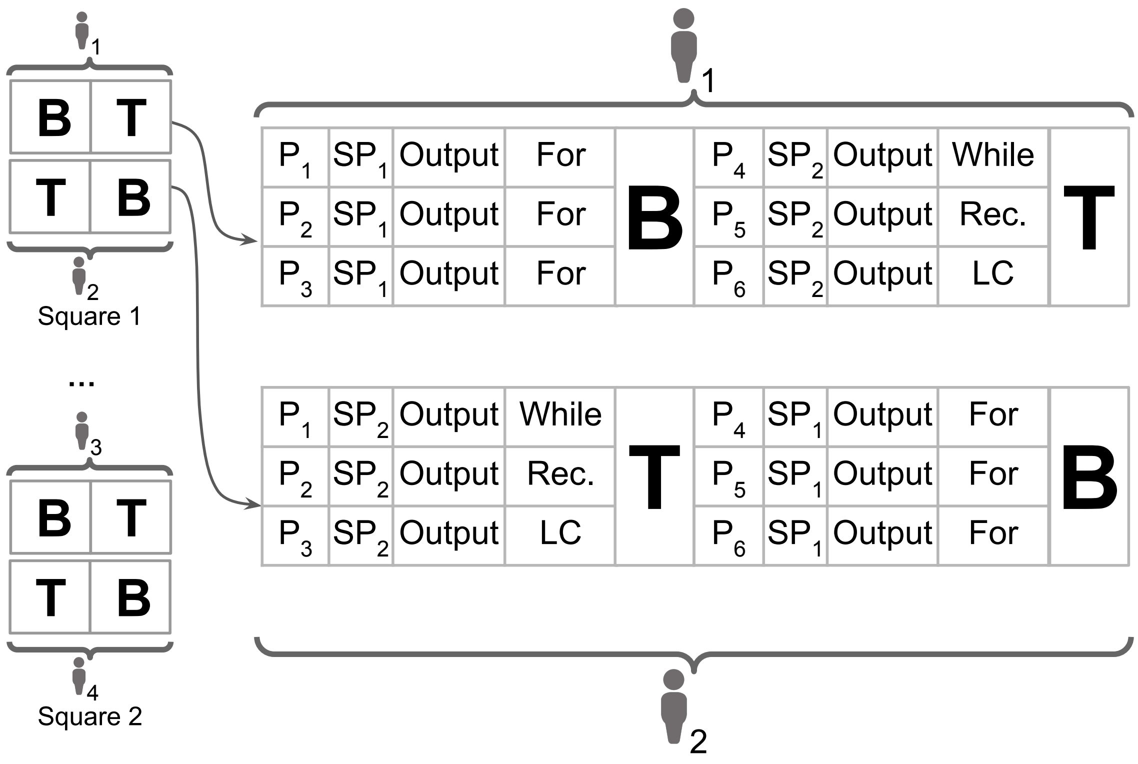
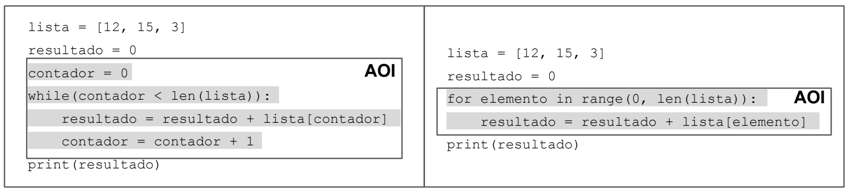


Figure 3.1: Experiment structure, Programs (P) are distributed in Program Sets (SP), where participants need to inform the code output (Output).

3.5 Programs

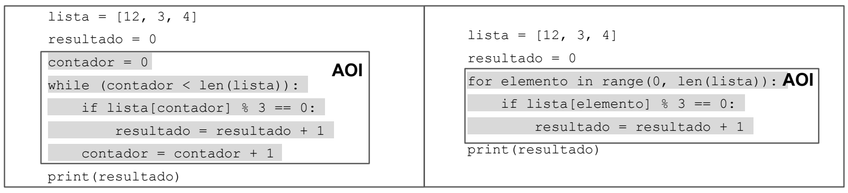
Code snippets were selected through an analysis carried out manually in repositories introducing programming tasks. The Code snippets used are shown in Figure 3.2. The main source of assignments was *GeeksforGeeks*, which is popular for learning and practicing programming. For the experiment, tasks with small and complete code snippets were selected, these snippets were adapted taking into account the needs of the experiment and the limitations of the camera. For each program, the participant needed to respond to the correct output in the open response model, i.e., no response options were provided. The methodology of



```
lista = [12, 15, 3]
resultado = 0
contador = 0
while(contador < len(lista)):
    resultado = resultado + lista[contador]
    contador = contador + 1
print(resultado)
```

```
lista = [12, 15, 3]
resultado = 0
for elemento in range(0, len(lista)):
    resultado = resultado + lista[elemento]
print(resultado)
```

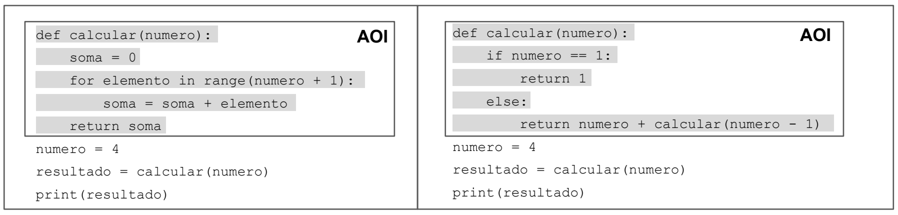
a) Sum list - while and for versions - CP1



```
lista = [12, 3, 4]
resultado = 0
contador = 0
while (contador < len(lista)):
    if lista[contador] % 3 == 0:
        resultado = resultado + 1
    contador = contador + 1
print(resultado)
```

```
lista = [12, 3, 4]
resultado = 0
for elemento in range(0, len(lista)):
    if lista[elemento] % 3 == 0:
        resultado = resultado + 1
print(resultado)
```

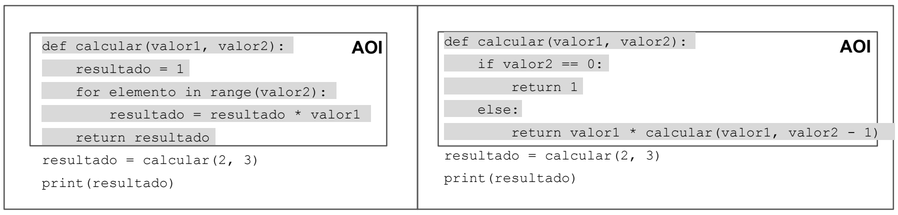
(b) Multiples of three - while and for versions - CP2



```
def calcular(numero):
    soma = 0
    for elemento in range(numero + 1):
        soma = soma + elemento
    return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

```
def calcular(numero):
    if numero == 1:
        return 1
    else:
        return numero + calcular(numero - 1)
numero = 4
resultado = calcular(numero)
print(resultado)
```

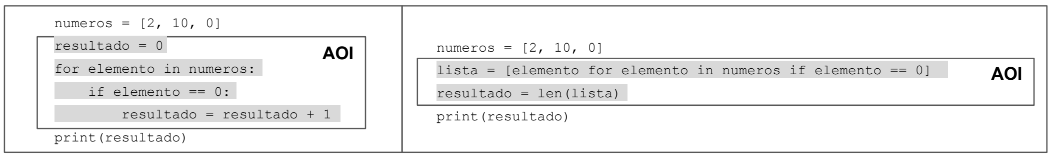
(c) Sum from 1 to n - for and Recursion versions - CP1



```
def calcular(valor1, valor2):
    resultado = 1
    for elemento in range(valor2):
        resultado = resultado * valor1
    return resultado
resultado = calcular(2, 3)
print(resultado)
```

```
def calcular(valor1, valor2):
    if valor2 == 0:
        return 1
    else:
        return valor1 * calcular(valor1, valor2 - 1)
resultado = calcular(2, 3)
print(resultado)
```

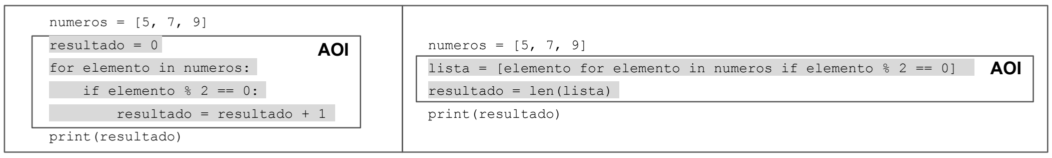
(d) Potentiation - for and Recursion versions - CP2



```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
    if elemento == 0:
        resultado = resultado + 1
print(resultado)
```

```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

(e) Quantity of 0 in the list - for and List Comprehension Versions - CP1



```
numeros = [5, 7, 9]
resultado = 0
for elemento in numeros:
    if elemento % 2 == 0:
        resultado = resultado + 1
print(resultado)
```

```
numeros = [5, 7, 9]
lista = [elemento for elemento in numeros if elemento % 2 == 0]
resultado = len(lista)
print(resultado)
```

(f) Number of pairs - for and List Comprehension Versions - CP2

Figure 3.2: Code snippets used.

providing information about the code, such as finding the output, is used by 70% of studies in the code understanding domain [47].

The evaluated programs had between 4-8 lines of code. The number of lines were limited so that they would fit completely on the screen. Programs have been checked to remove syntax errors. They were also organized in the PEP8 standard using the Python Syntax Checker PEP8 tool. The Consolas font style was used, size 12, line spacing 1.5 inches and eight blank spaces for indentation. Simple constructions that commonly occur in many languages were used.

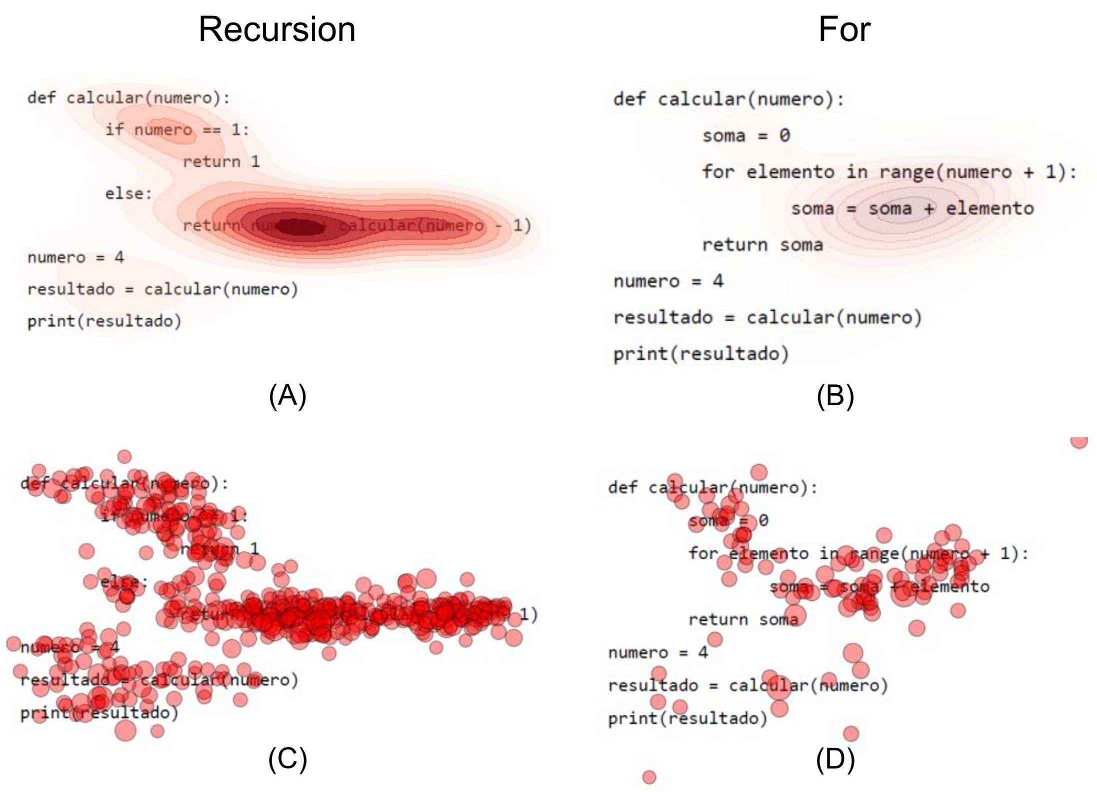
3.6 Eye tracking system

For the experiment, the Tobii Eye Tracker 4C equipment was used, which has a sample rate of 90 Hz. Eye tracking calibration followed the device driver's standard procedure with five points. The eye tracking camera was mounted on a laptop screen with a resolution of 1366 x 720 pixels, height of 30.9 cm and width of 17.4 cm, at a distance of 50-60 cm from the participant. Each task was presented in an image in full screen mode, but an Integrated Development Environment (IDE) was not used, nor was the number of lines. An accuracy error of 0.7 degrees was calculated, which translates into 0.6 lines of printing on the screen, considering the font size and line spacing. The line spacings were designed to be large enough to overcome the accuracy limitations of the eye tracker. To process the data, a Python script was used, which made it possible to collect and analyze the metrics.

3.7 Planning of the experiment

To carry out the experiment, the programs, characterization form, consent form and a questionnaire for a semi-structured interview were used. To evaluate the programs that would be used, code excerpts were tested with different levels of difficulty. Fixed standards for code font size, font style, line spacing and indentation were also used. The questions on the forms and questionnaire were also evaluated. The pilot study is described in more detail in the next sections.

3.7.1 Sum from one to *n*, Recursion and `for` versions



```
def calcular(numero):
    if numero == 1:
        return 1
    else:
        return numero + calcular(numero - 1)
numero = 4
resultado = calcular(numero)
print(resultado)
```

```
def calcular(numero):
    soma = 0
    for elemento in range(numero + 1):
        soma = soma + elemento
    return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

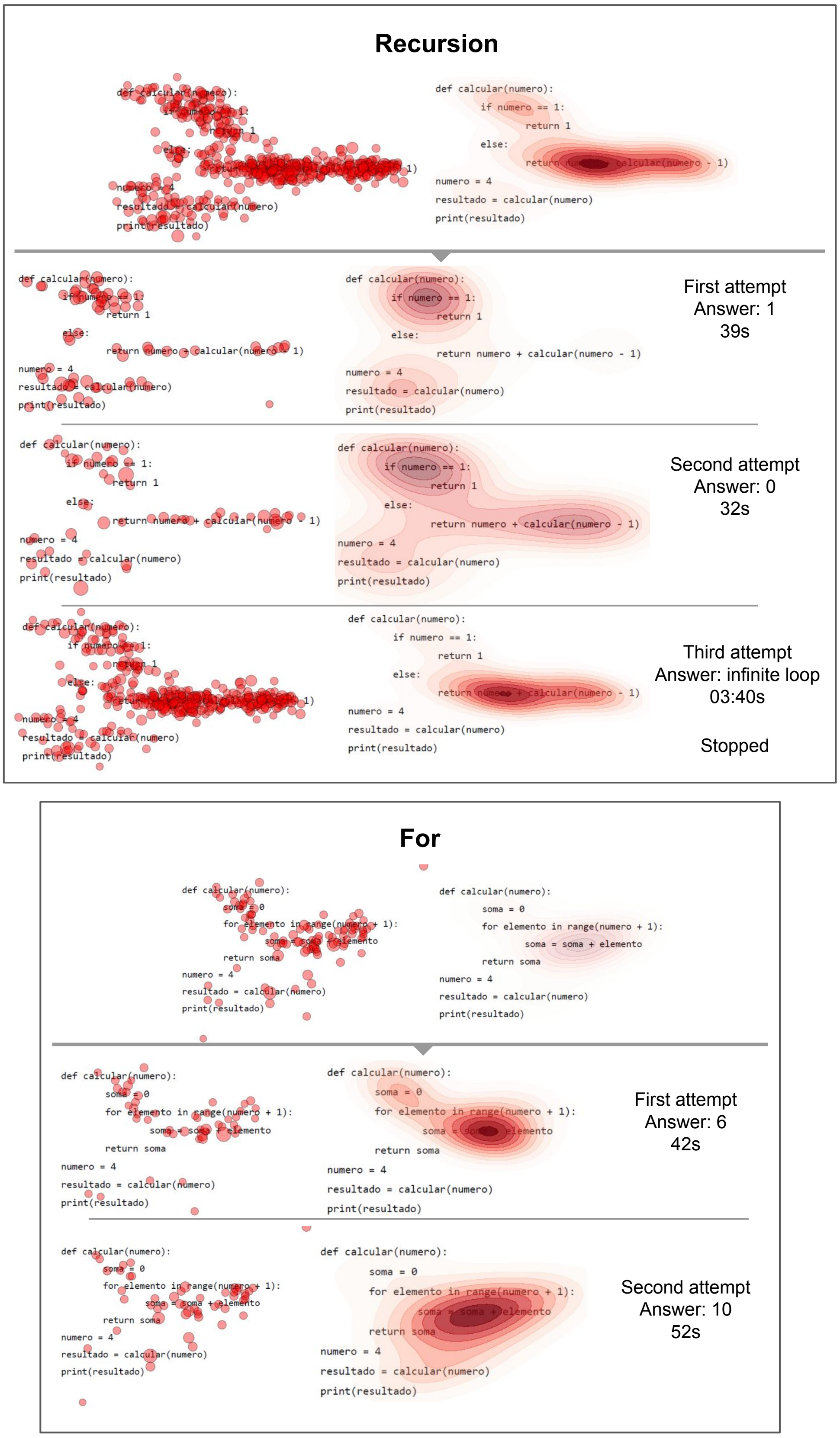
Figure 3.3: Comparison between the two code snippets Recursion and `for` and their respective heat maps and fixations for the task *Sum from one to n*.

In Figure 3.3, the task *Sum of one to n* is presented, in the Recursion and `for` versions, and their respective heat and fixation maps generated from the data of the eye tracking. The heat map allows intuitive and informative visualization of participants' attention behavior. It highlights the areas of the stimulus that most attract attention and those that are least explored. With this, it is possible to identify which elements of the code are most interesting to participants and which regions or elements may go unnoticed. In the heat map, the intensity of red varies according to the number of fixations and their duration, making it possible to make comparisons between versions of the same code that receive attention differently. The fixing map presents a visualization of the distribution of fixings in the code. Instead of showing the intensity of fixations with colors, as in a heat map, the fixation map indicates the exact locations where participants' eyes fixated. They help identify specific points of interest or areas of focus in code stimuli that may go unnoticed in heatmaps.

In the Recursion version, assessed by the participant as difficult, it took 4min52s and three attempts, but the participant ended up giving up. In the heat map, a greater visual effort is identified in the region of the Recursion call, in the recursive step. This can be confirmed in the interview, in which the participant mentioned difficulties in calling the Recursion and in its loop: "*The issue of recursion, I didn't understand how I was adding the value assigned to the number with the calculation of the number - 1, for me, it would make a loop, just adding the number minus 1.*" In this case, a greater intensity of red in the recursive step region is associated with difficulty for the participant in solving the task. In this way, the participant looks at this region repeatedly, which contributes to an increase in the time and number of fixations and gives clues about the reasons for the participant's withdrawal.

In Figure 3.4, left side, three attempts by the participant are shown with their respective visual behaviors for the task with Recursion. In the first attempt, 39s were spent. It is possible to observe especially from the heat map that attention is focused in two main places: in the base case and in the function call at the end of the code. The participant submitted the answer incorrectly. The answer 10 was expected and the participant answered 1. In the second attempt, 32s were spent and, in addition to the base case, visual attention is more intense on the function call in the middle of the code, in the recursive step, resulting in the submission of the answer incorrect 0. In the third submission, 3min40s were spent. Visual attention focuses mainly on the function call in the middle of the code, in the recursive step, resulting in an incorrect answer in which the participant responded that it was an infinite loop. Finally, the participant gave up. This difficulty may be related to the fact that it is a function that is called repeatedly, requiring dynamic memorization and having a more abstract resolution format. This could justify the difficulty encountered in the recursive step region and in the function loop.

In Figure 3.5, two code snippets are presented, one version with `for` and another with Recursion, in addition to the participant's preference and the reason for their preference. It is possible to observe that the version with Recursion was preferred by both participants, mainly because it seems clearer. The following reasons were presented: "*I think recursion makes execution cleaner*" and "*It's simpler for me to understand, the same thing as before, the definition within the range I I did not remember. Even though it has recursion*". However, when analyzing performance, in the version with Recursion, the time spent was longer, and

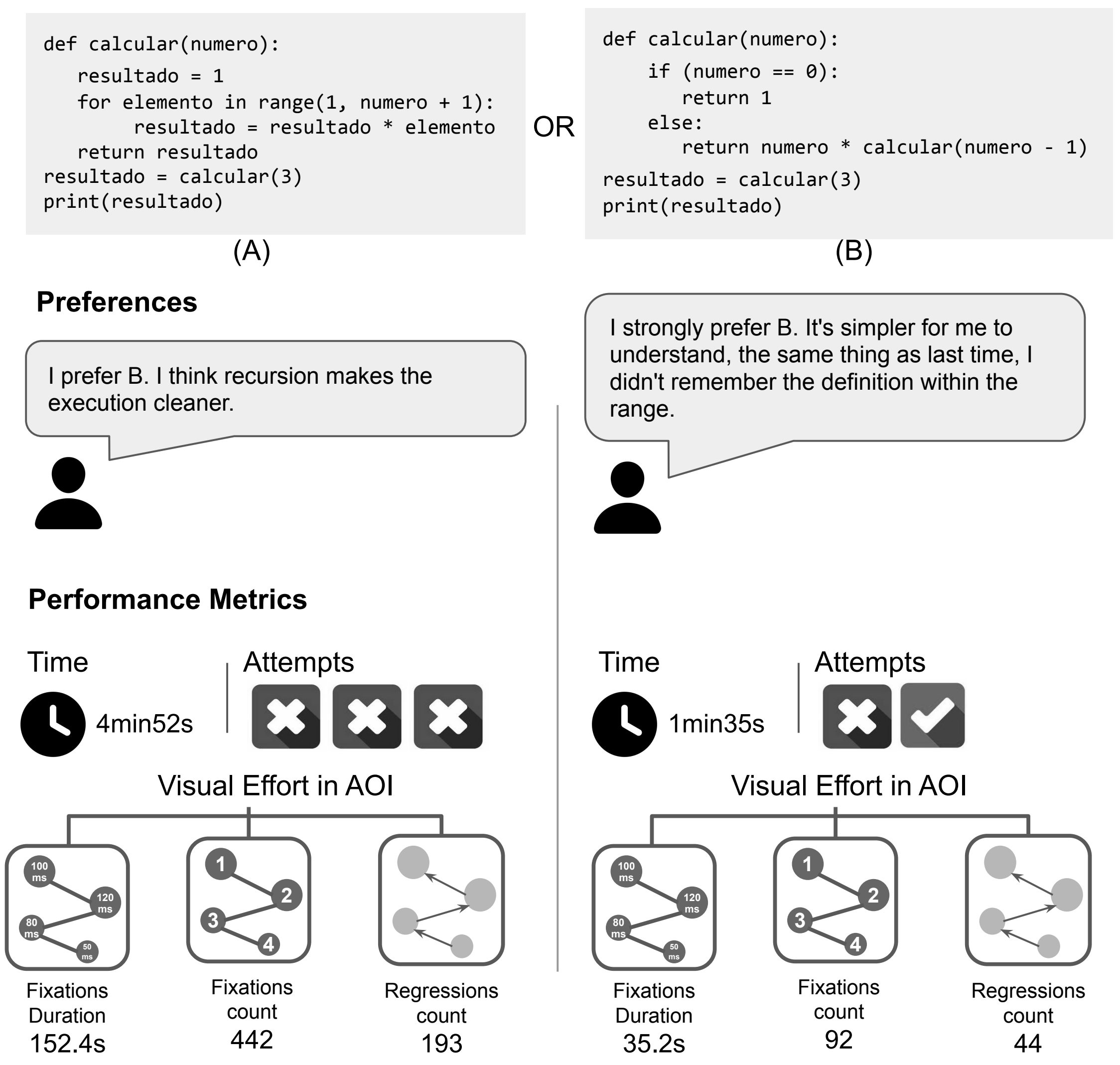


```
def calcular(numero):
        if numero == 1:
                return 1
        else:
                return numero + calcular(numero - 1)
numero = 4
resultado = calcular(numero)
print(resultado)
```

```
def calcular(numero):
        soma = 0
        for elemento in range(numero + 1):
                soma = soma + elemento
        return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 3.4: Task submissions in the Recursion and `for` versions.

there was a dropout. Although the preferred version was Recursion, the version with `for` proved to be better in evaluating the metrics, thus, there was no agreement between the preference and the participants' performance in this task.



```
def calcular(numero):
   resultado = 1
   for elemento in range(1, numero + 1):
        resultado = resultado * elemento
   return resultado
resultado = calcular(3)
print(resultado)
```

```
def calcular(numero):
    if (numero == 0):
       return 1
    else:
       return numero * calcular(numero - 1)
resultado = calcular(3)
print(resultado)
```

I prefer B. I think recursion makes the execution cleaner.

I strongly prefer B. It's simpler for me to understand, the same thing as last time, I didn't remember the definition within the range.

Figure 3.5: Comparison between preference and performance of the `for` and Recursion versions.

Through the analysis of heat maps and fixations, relating them to the number of attempts, it was possible to notice some effects in terms of visual attention. For example, the visualization of some areas in the code are essential for its resolution, such as the base case and the recursive step, and it is possible to see that the participant is aware of these locations by the intensity of the red in the heat map. However, even looking at these locations, the participant gave up.

The `for` version, whose difficulty was rated as Neutral, took 1min35s and two attempts to solve. In the heat map, Figure 3.3, visual attention is identified in the region where the sum is performed, the instruction within `for`. The participant mentioned difficulties regarding `range()`: "*the range, it is a number + 1, I just put it as a number, I forgot to add + 1.*"

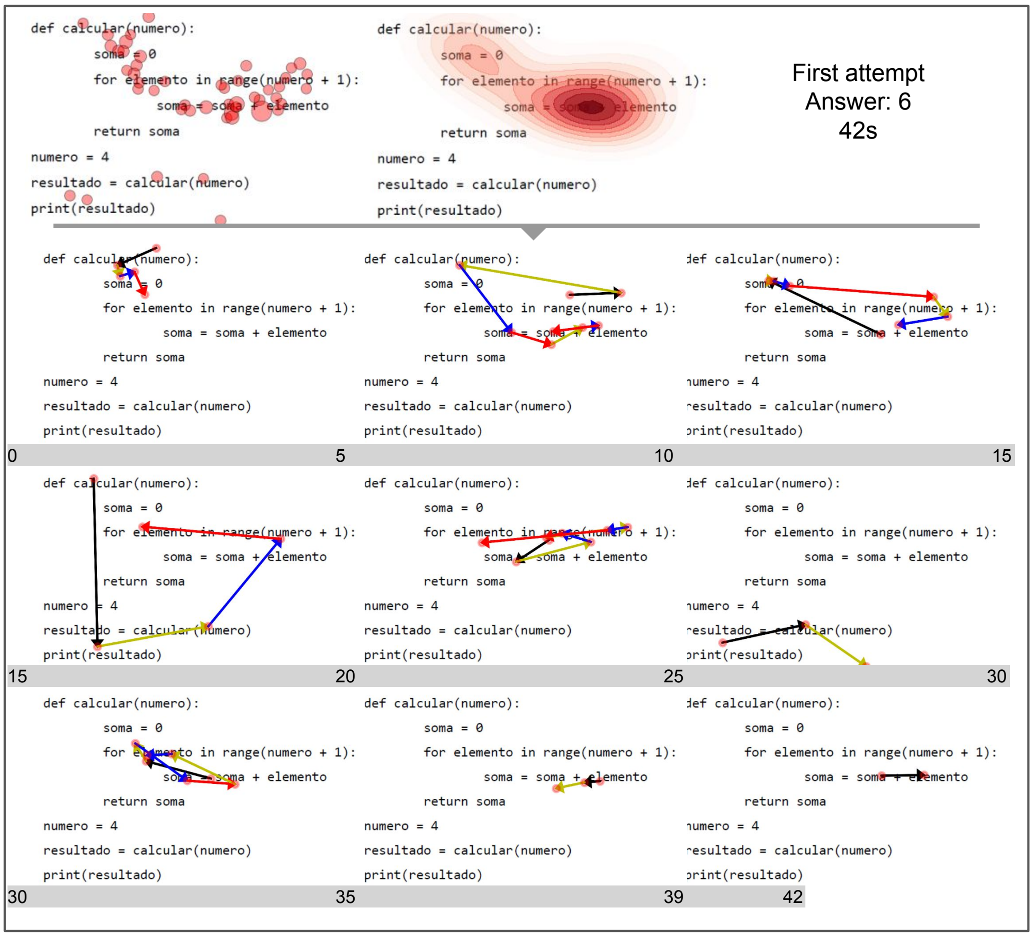
As for submissions, in Figure 3.4, in version `for`, 42s were spent on the first submission. It is possible to observe that visual attention is concentrated in two main places, in the declaration of the variable `sum` and in the region where the sum is performed, the instruction within `for`. For this task, the participant gave the incorrect answer 6 when 10 was expected. In the second submission, 52s were spent. Visual attention appears stronger in the region where the sum is performed, the instruction within `for`, but now it is expanded towards the function call and `range()`, and the participant responded correctly 10.

In Figure 3.6, *Gaze Transitions* of the `for` version are shown, which are gaze transitions in the task. As the difficulty in *range()* was mentioned, it is possible to observe that in the second submission, the participant seems to pay more attention to the mentioned part of the code, with several small transitions going back and forth to the instruction inside `for`, especially between 5s and 20s.

3.7.2 Task *Multiple of three*, versions `while` and `for`

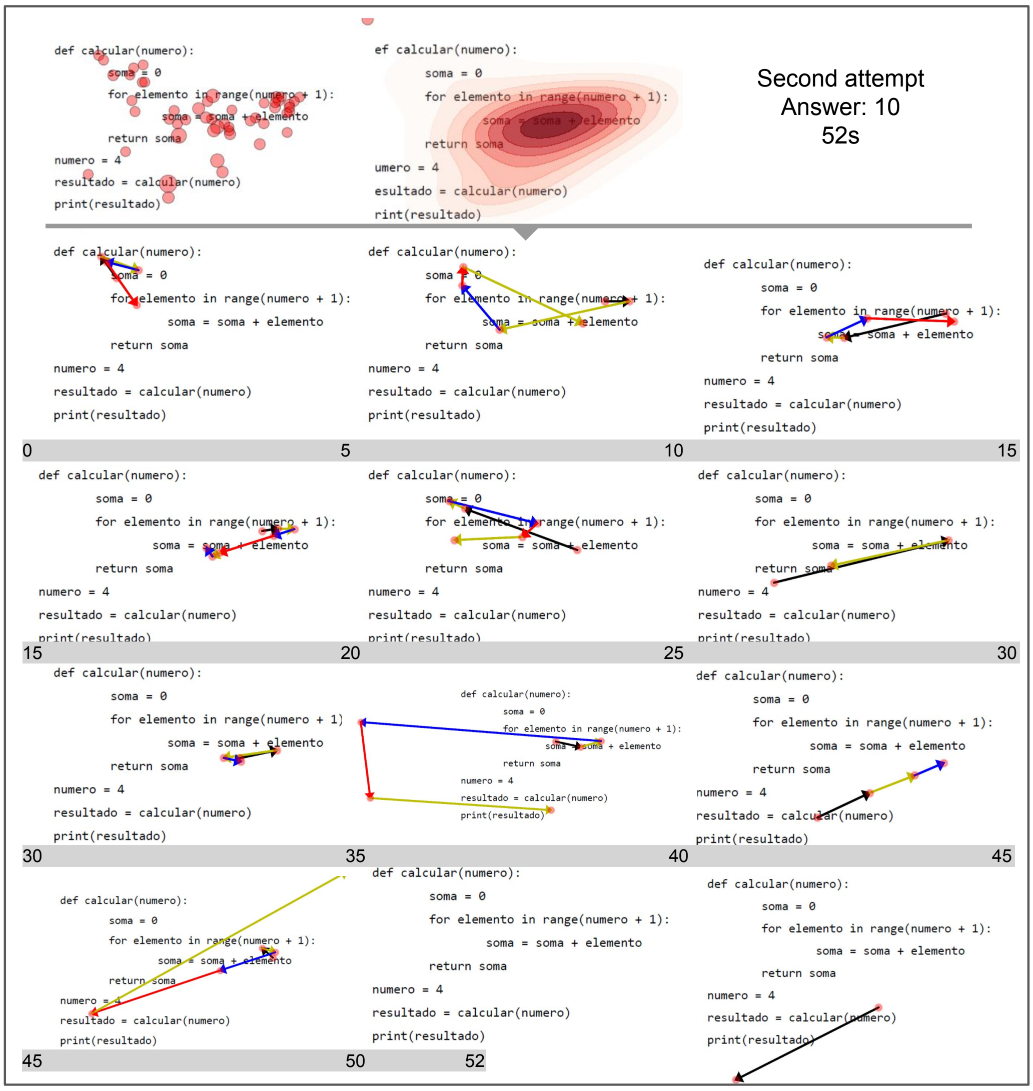
In Figure 3.8, the task *Multiple of three* is presented, in versions `while` and `for`, and their respective heat and fixation maps generated from the eye tracking data. In the `for` version, it took 37s and one attempt. In the heatmap, greater visual effort is identified in two main regions, in the region of the list declaration and in the module of 3 in the comparison within `if`. This can be confirmed in the interview, in which the participant mentioned the strategy used to solve the task: "*Reading the code, it asks for the result, I went back to the result up there, I entered `for` and the rest of the division by 3, the only one in the list that divided by 3 does not give 0 is the last one, result plus 1.*" In this case, a greater intensity of red is associated with the main regions used by the participant to solve the task.

In Figure 3.9, right hand side, *Gaze Transitions* of the task with version `for` are shown. As mentioned by the participant in the interview about the strategy used to solve the task, it is possible to observe several small transitions in the `if` line, and long transitions between the `if` and list declaration regions.



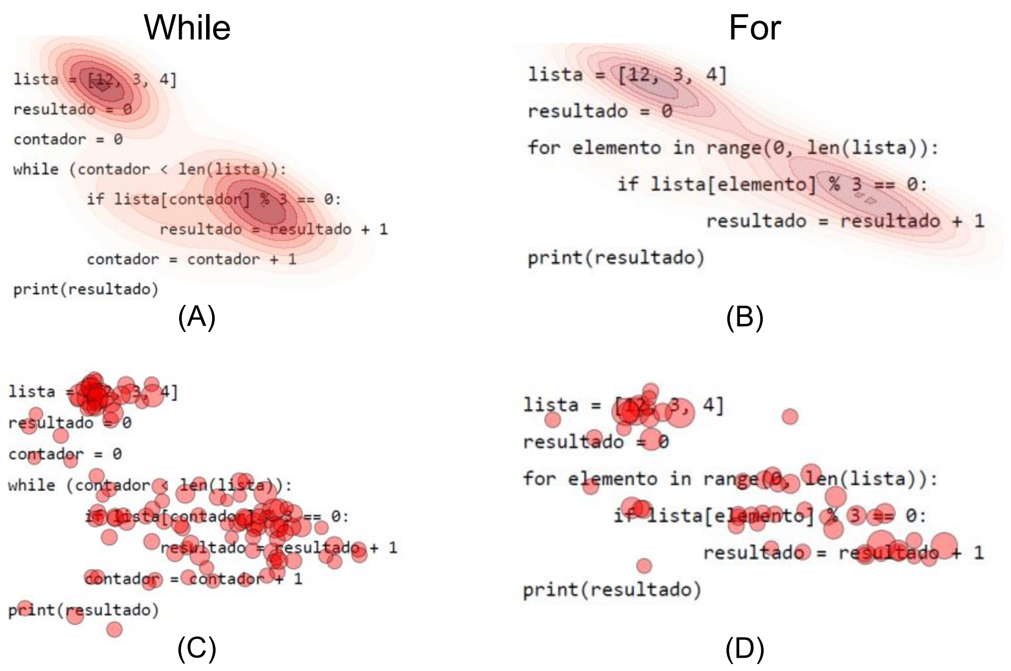
```
def calcular(numero):
        soma = 0
        for elemento in range(numero + 1):
                soma = soma + elemento
        return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 3.6: Gaze Transitions of task submissions *Sum from one to n* of version `for`, First attempt.



```
def calcular(numero):
        soma = 0
        for elemento in range(numero + 1):
                soma = soma + elemento
        return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 3.7: Gaze Transitions of task submissions *Sum from one to n* of version `for`, First attempt, Second attempt.



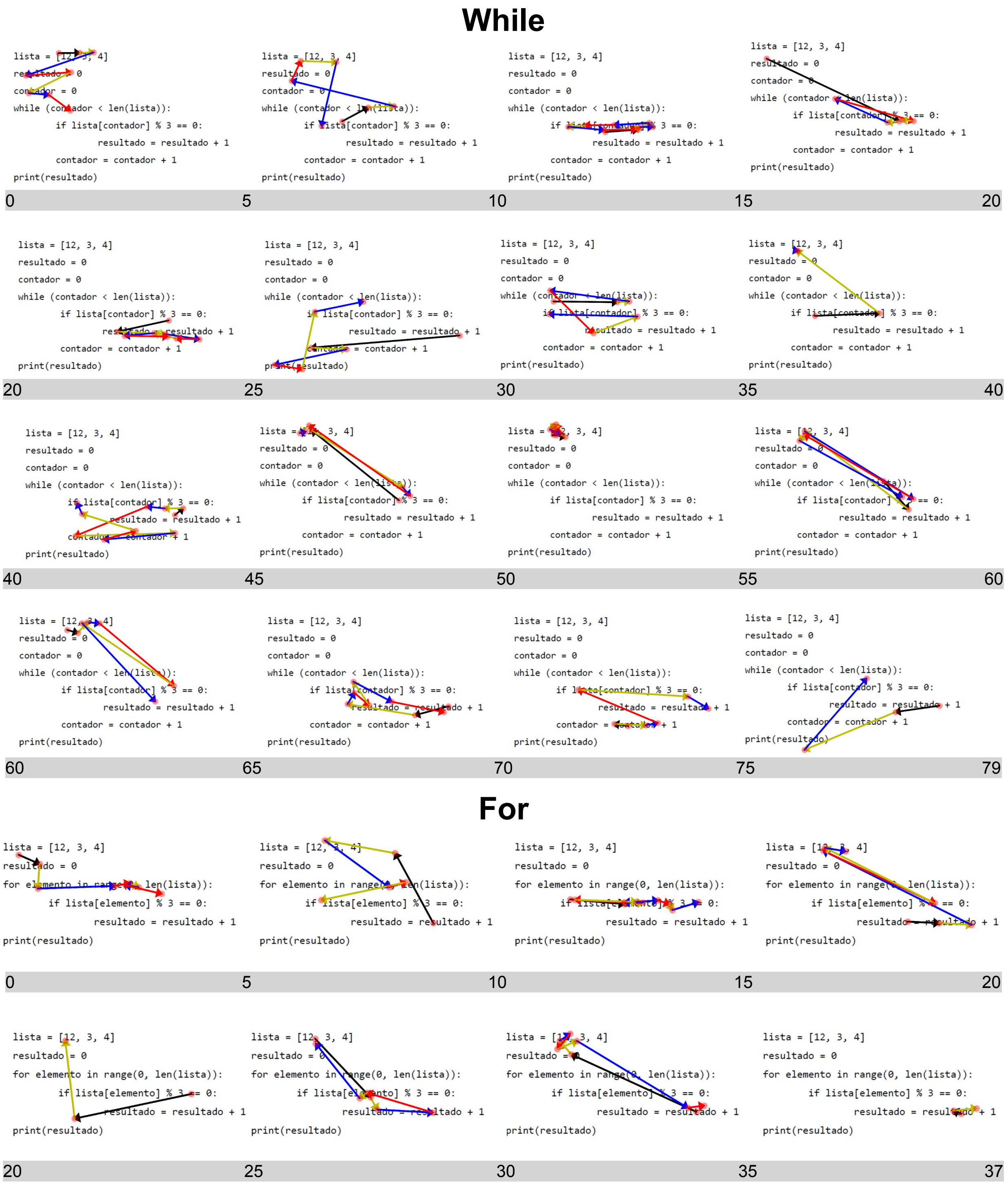
```
lista = [12, 3, 4]
resultado = 0
contador = 0
while (contador < len(lista)):
        if lista[contador] % 3 == 0:
                resultado = resultado + 1
        contador = contador + 1
print(resultado)
```

```
lista = [12, 3, 4]
resultado = 0
for elemento in range(0, len(lista)):
        if lista[elemento] % 3 == 0:
                resultado = resultado + 1
print(resultado)
```

Figure 3.8: Comparison between the two code snippets `while` and `for` and their respective heat maps and fixations for the task *Amount of multiples of three in a list.*

In Figure 3.10, two code snippets are presented, one version with `while` and another with `for`, in addition to the participant's preference and the reason for their preference. It is possible to observe that there was no unanimity in preference. The `for` version was preferred by the participant, mainly because it looks cleaner, less code, fewer variables and allocation. The following reasons were presented: "*I think the code is cleaner, less code, easier to execute. Fewer characters to read, this also saves a lot on reading the code.*", when asked why the preference for a cleaner code, the following answer was given "*Because I think it makes it a little easier too in terms of visualization, as well as execution must also be much faster than executing code with a greater number of variables, you will have to do more allocation, something like that.*" When analyzing performance, on average, the tasks with version `for` required less time to be resolved. For this comparison, there was an agreement between the participants' preferences and their respective performances.

In the `while` version, Figure 3.8 left side, 1min:19s and one attempt were spent. In the heat map, greater visual effort is also identified in two main regions, in the region of the list declaration and in the module of 3 in the comparison within `if`. This can be confirmed in the interview, in which the participant mentioned the strategy used to solve the task: "*Trying*



```
lista = [12, 3, 4]
resultado = 0
contador = 0
while (contador < len(lista)):
        if lista[contador] % 3 == 0:
                resultado = resultado + 1
        contador = contador + 1
print(resultado)
```

```
lista = [12, 3, 4]
resultado = 0
for elemento in range(0, len(lista)):
        if lista[elemento] % 3 == 0:
                resultado = resultado + 1
print(resultado)
```

Figure 3.9: Gaze Transitions of task submissions *Amount of multiples of three in a list*, comparison between the two code snippets `while` and `for`.

to do the module, the only one that didn't have module 3 was 4, so there would be his module left , which adds another 1 and the result would be 2. At most it would give 3, the while would give 3 loops." Also in this case, a greater intensity of red is associated with the main regions used by the participant to solve the task.

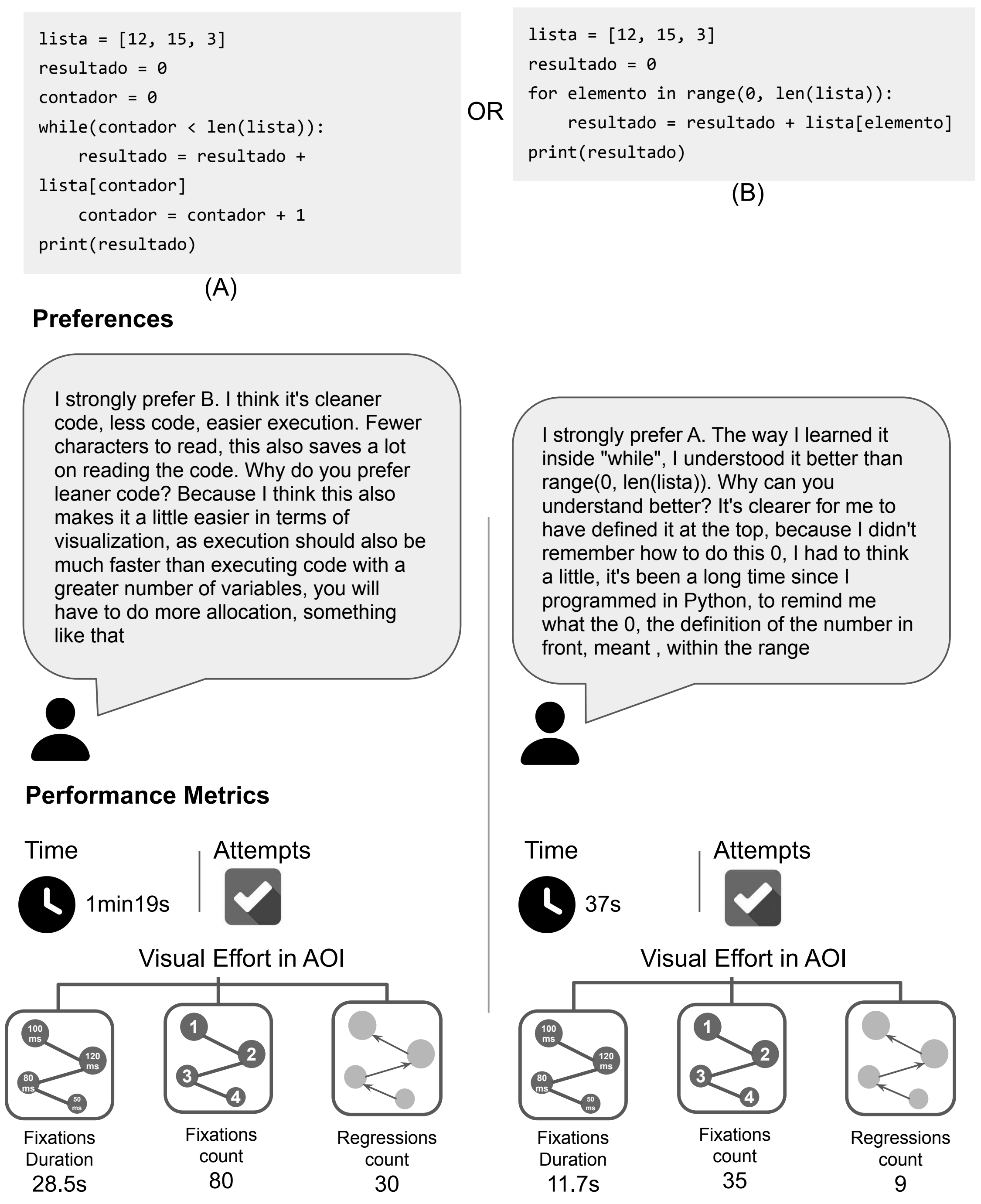
In Figure 3.9, left side, *Gaze Transitions* of the task with version `while` are shown. As mentioned by the participant in the interview about the strategy used to solve the task, it is also possible to observe several small transitions in the `if` region, and longer transitions between the `if` and declaration regions.

In Figure 3.10, the participant who preferred the `while` version mentioned difficulty remembering how `range` works in `for`. Thus, he presented the following reasons for his preference "*The way I learned within while, I understood it better than range(0, len(lista)).*", when asked why he was able to understand better, the following answer was given "*It's clearer for me to have defined it above, because I didn't remember how to do this 0, I had to think a little, it's been a long time since I programmed in Python, so it reminds me what does 0, a definition of the number in front meant, within the range.*" When analyzing performance, on average, tasks with version `while` needed more time to solve. For this comparison, there was also an agreement between the participants' preferences and their respective performances.

3.7.3 Task *Quantity of 0 in list*, LC and `for` versions

In Figure 3.11, the task *Quantity of 0 in the list* is presented, in the CL and `for` versions, and their respective heat and fixation maps generated from the data of the eye tracking. In the CL version, 41s and one attempt were spent. In the heat map, greater visual effort is identified in the `len(lista)` and `if` regions. This can be confirmed in the interview, in which the participant mentioned the difficulty: "*Easy. Because the condition there is directing directly to this condition there, it only returns if it is 0, which in this case is the only element within the list of numbers.*" In this case, a greater intensity of red is associated with the main region used by the participant to solve the task.

In Figure 3.12, left side, *Gaze Transitions* of the task with the CL version are shown. As mentioned by the participant in the interview regarding the difficulty in solving the task, it is possible to observe several small transitions in the `len(lista)` region, and long transitions



```
lista = [12, 15, 3]
resultado = 0
contador = 0
while(contador < len(lista)):
    resultado = resultado +
lista[contador]
    contador = contador + 1
print(resultado)
```

(A)

OR

```
lista = [12, 15, 3]
resultado = 0
for elemento in range(0, len(lista)):
    resultado = resultado + lista[elemento]
print(resultado)
```

(B)

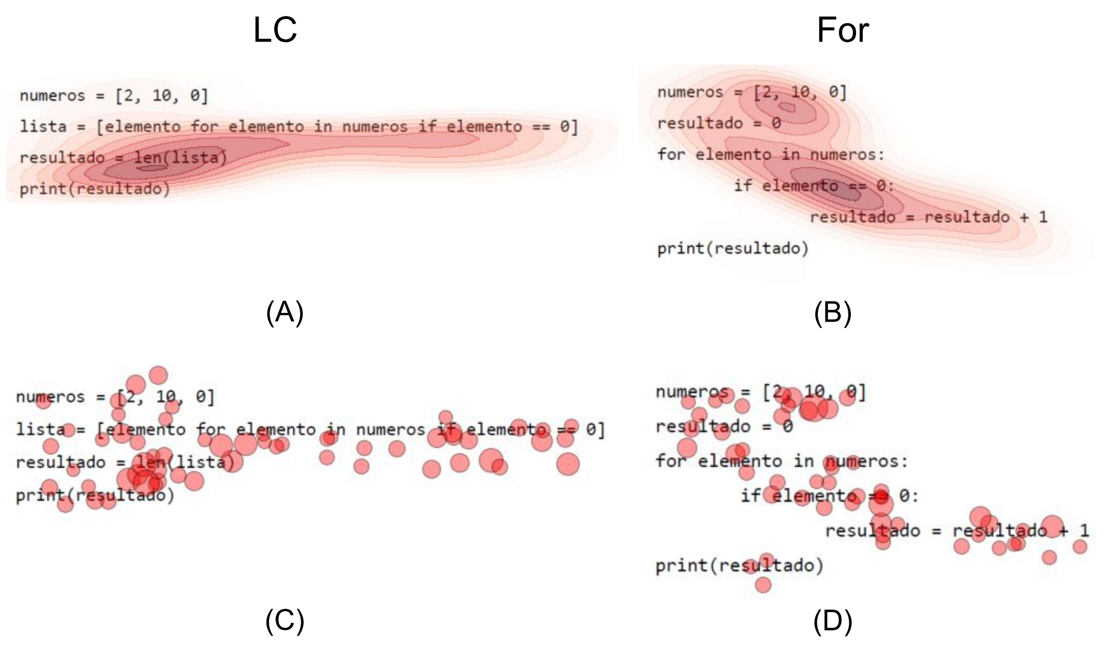
Preferences

I strongly prefer B. I think it's cleaner code, less code, easier execution. Fewer characters to read, this also saves a lot on reading the code. Why do you prefer leaner code? Because I think this also makes it a little easier in terms of visualization, as execution should also be much faster than executing code with a greater number of variables, you will have to do more allocation, something like that

I strongly prefer A. The way I learned it inside "while", I understood it better than range(0, len(lista)). Why can you understand better? It's clearer for me to have defined it at the top, because I didn't remember how to do this 0, I had to think a little, it's been a long time since I programmed in Python, to remind me what the 0, the definition of the number in front, meant , within the range

Performance Metrics

Figure 3.10: Comparison between preference and performance of code versions `while` and `for`.



```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
    if elemento == 0:
        resultado = resultado + 1
print(resultado)
```

```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
    if elemento == 0:
        resultado = resultado + 1
print(resultado)
```

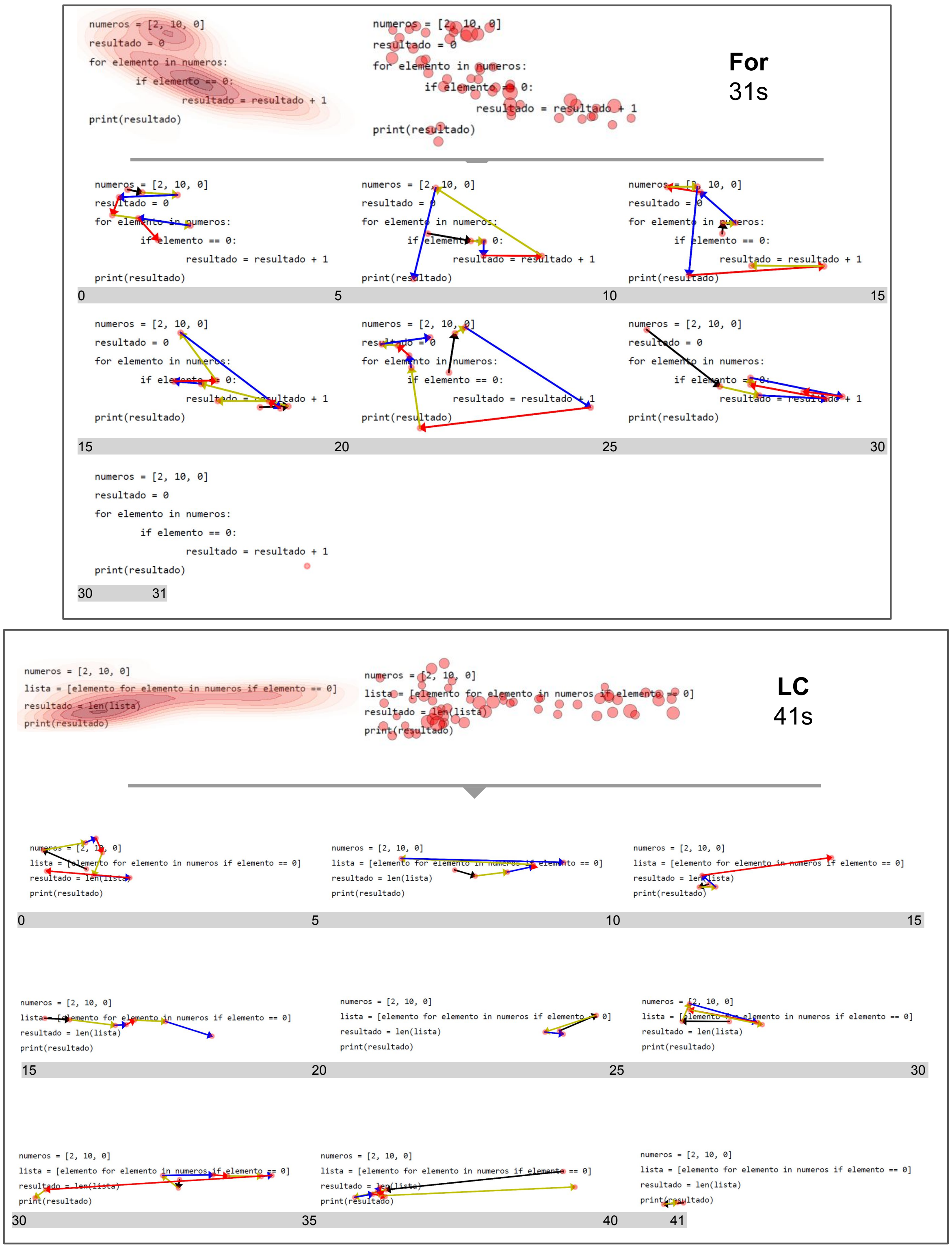
Figure 3.11: Comparison between the two code snippets CL and `for` and their respective heat maps and fixtures for the task *Quantity of 0 in the list*.

between this region and the `if` region.

In Figure 3.13, the metrics time, attempts and visual effort in the AOI are presented, related to the LC version and `for`. When analyzing performance, it was observed that the task implemented with the `for` loop showed a reduction in time, number of fixations and duration of fixations to be solved, compared to the LC version.

In version `for`, Figure 3.11, 31s and one attempt were spent. In the heatmap, greater visual effort is identified in two main regions, in the list declaration region and in the comparison within `if`. This can be confirmed in the interview, in which the participant mentioned the strategy used: "*I read the entire code and it asks for the result, but it only enters if if the result is 0, it is only 0 in the last position of the list.*" Also in this case, a greater intensity of red is associated with the main regions used by the participant to solve the task.

In Figure 3.12, right side, *Gaze Transitions* of the task with version `for` are shown. As mentioned by the participant in the interview regarding the strategy for solving the task, it is possible to observe several small transitions in the `if` region, and longer transitions between the `if` region and the list declaration.



```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
        if elemento == 0:
                resultado = resultado + 1
print(resultado)
```

```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

Figure 3.12: Gaze Transitions from task submissions *Amount of 0 in list*, comparison between the two code snippets CL and `for`.

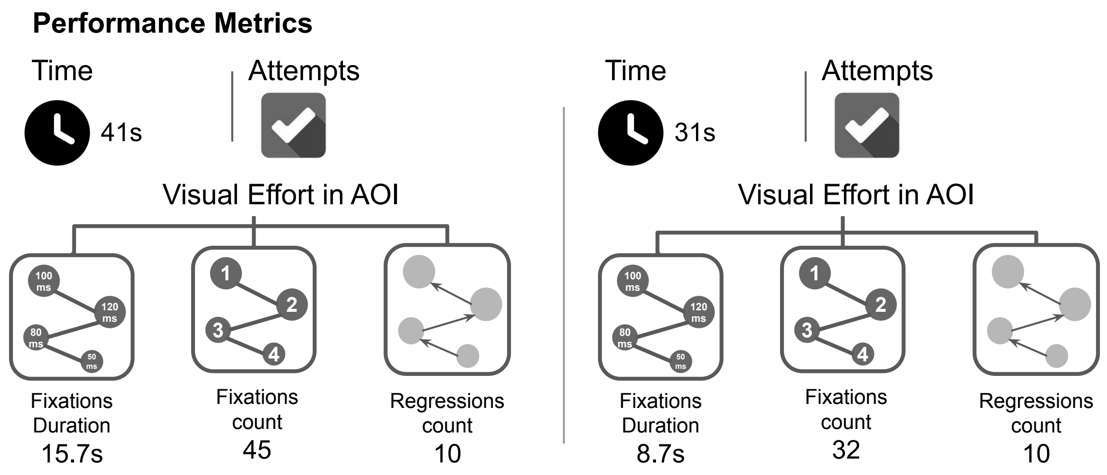


Figure 3.13: Performance comparison in LC and `for` code versions of the task *Quantity of 0 in the list*.

Chapter 4

Evaluation

In this section, we describe the evaluation of our approach. In Section 4.1, we present the goal of our evaluation along with our research questions and metrics. In Section 4.2, we present the main results obtained and discuss our results. In Section 4.3, we answer our research questions and finally, in Section 4.4, we describe the threats to validity.

4.1 Definition

In this section, we present the objective of our study following the Goal-Question-Metric approach [3]. **We compare** programs with Recursion techniques, repetition structure and LC **with the purpose of** understanding how these techniques impact code comprehension **in relation to** time, number of attempts and visual metrics (duration fixation count, fixation count, regression count) **from the point of view of** Python novices **in the context of** tasks adapted from introductory programming courses. Therefore, this work seeks to contribute to a deeper understanding of the impact of different ways of structuring the code on the understanding of novices. In particular, this work seeks to carry out a controlled experiment with Python novices seeking to answer five research questions. For each of the questions, the null hypothesis is that there is no difference in terms of the metrics used, between the versions of the programs with `for` and the versions with `while`, Recursion and LC.

We address the following Research Questions (RQs):

- **RQ$_1$ What is the impact of code structures on the time to solve the task?** Similarly to previous studies, which used the task resolution time metric to investigate code understanding [16; 17; 19; 60], to answer this question, the time the participant spent reading the code was measured, understand, and report the correct output of the program, given input into the complete code itself. Additionally, the time that novices spend in specific areas within the code, the AOIs, is measured. Time was measured using the eye tracking camera system.

- **RQ$_2$ What is the impact of code structures on the number of attempts made to solve the task?** Based on previous studies, which used the correctness of the task as a way of investigating code understanding [16], to answer this question, the number of submissions made by the participants was counted, from the beginning of the task until the moment in which they responded correctly to exit the program or chose to give up the task.

- **RQ$_3$ What is the impact of code structures on the duration of fixations?** The position and duration of fixations have been associated with the focus of attention [14]. Similar to previous studies [17; 16], to answer this question, the duration of fixations in the full code and in the AOI was measured. In Figure 2.5(b), an example of the duration of fixations is presented.

- **RQ$_4$ What is the impact of code structures on the number of fixations?** A high number of fixations is associated with a greater visual effort to answer a question [59]. To answer this question, the number of fixations in the complete code and in the AOI was measured. In Figure 2.5(a), an example of the fixation count is shown.

- **RQ$_5$ What is the impact of code structures on the number of regressions?** Returning eye movements to stimuli are called regressions [11]. Regressions can be used as a metric for visual effort [57]. Therefore, to answer this question, the number of regressions in the complete code and in the AOI was measured. In Figure 2.5(c), an example of regression counting is presented.

4.2 Results and Discussion

In this section, we present the results obtained in the study carried out with 32 subjects. The tasks were analyzed with a main focus on AOI. Pilot studies were not considered for analysis of quantitative results. The experiment was conducted with 34 participants. In the end, the participant who had the least experience was removed, resulting in the removal of the Latin Square, ending with 32 participants. The experiment followed the same methodology used in the pilot study. The pilot study is described in more detail in Section 3.1.

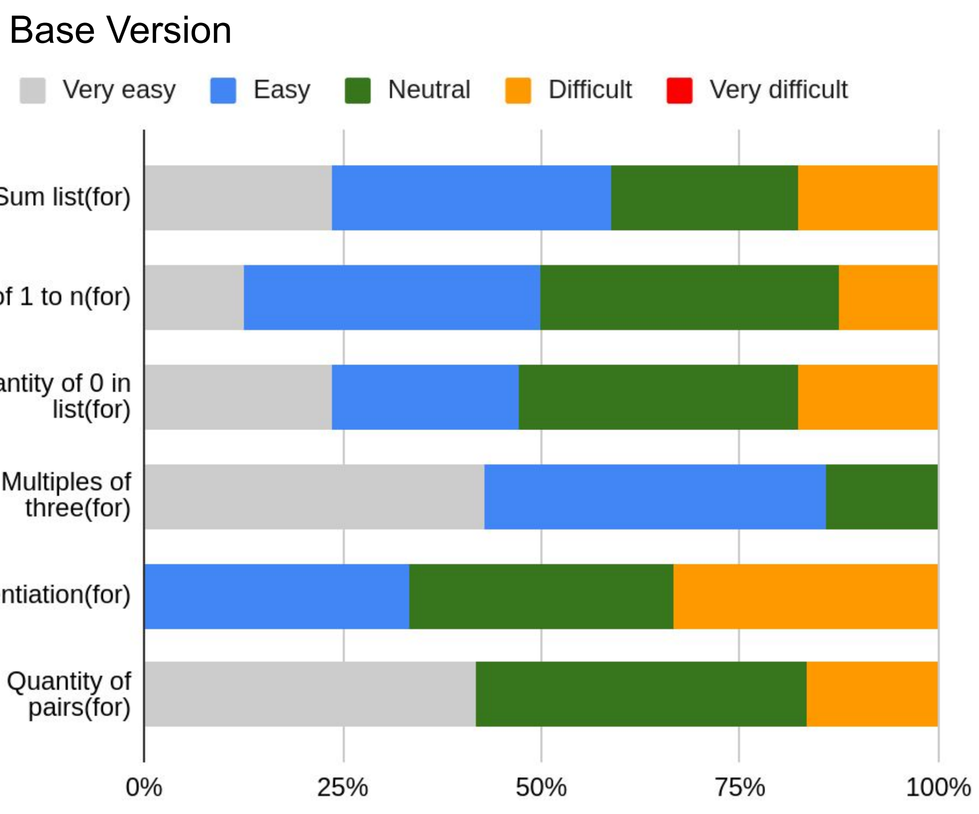
4.2.1 Task *Sum list*, `while` and `for` versions

In Figure 4.2, we depict the task *Sum from one to n*, in the `while` and `for` versions, and their respective heatmap and fixation maps generated from the data of the eye tracking. The heat map allows intuitive and informative visualization of participants' attention behavior. It highlights the areas of the stimulus that most attract attention and those that are least explored. With this, it is possible to identify which elements of the code are most interesting to participants and which regions or elements may go unnoticed. In the heat map, the intensity of red varies according to the number of fixations and their duration, making it possible to make comparisons between versions of the same code that receive attention differently. The fixing map presents a visualization of the distribution of fixings in the code. Instead of showing the intensity of fixations with colors, as in a heat map, the fixation map indicates the exact locations where participants' eyes fixated. They help identify specific points of interest or areas of focus in code stimuli that may go unnoticed in heatmaps. In the task with While version, the participant used 198.99s, 2 attempts, 207 fixations, 61.01s in fixation duration, 78 regressions and gave up. In the task with For version, the participant used 42.74s, 1 attempt, 35 fixations, 9.51s in fixation duration, 14 regressions and solved the task.

It is possible to observe that the most interesting regions of the code for the participant who performed the task with the While version and gave up, are mainly in the regions where result, counter and list[counter] are located. This can also be identified in the participant's comment regarding their strategy to solve the task:"*There was the list, right, as long as the counter is smaller than the size of the list, result + list, the counter I think it was, in that part I got lost a little bit, when counter arrived + 1, which was just to avoid repeating, I just took it and went to the result, + list (counter), I thought about the size of the list, but I thought 0, 1, 2, I hadn't realized it was the size, 0, 1, 2, I thought 2. Then on the other attempt, I went by the number of numbers, by the size.*" It is possible to observe that the most interesting regions of the code for the participant who performed the task with the For version are mainly in the regions where list and len[lista] are located. This can also be identified in the participant's comment regarding his strategy to solve the task: "*I saw that it was a list with three numbers, and the result was equal to 0 first, then when I saw the (for element in range), when I saw that the result would be the result that was initially + the list elements in this case, 12, 15.3, as I know that 12+15+3 is 30, I said 30.*"

To analyze the participants' perception of the difficulty of the programs, a five-point scale was used, shown in Figure 4.1. For each program, there were two versions and participants had to rate the difficulty they encountered in solving it on a scale of very easy, easy, neutral, difficult, or very difficult to solve. In general, it was possible to observe that the participants' perception of the difficulty between the programs presented was that the `for` version seemed easier than the other versions.

Some of the main reasons given by the subjects for preferring the `for` version were less variable, it does not use a counter and there is no need to increment the counter, "*because as it already has a list with a fixed size, which is passed within* `for`*, automatically won't need the counter, so it will reduce the number of variables, so much so that it will be a global variable, right, I won't have a counter as a global variable, right, and I won't have to be incrementing it locally within* `for` *, it's smaller, I have the feeling that the complexity is smaller and easier to understand*", "*Because it's a simpler code to write*", a cleaner and smaller code, "*I'm very used to* `for` *than with* `while`*, because I think it's a cleaner code, that doesn't need to break, doesn't need to use a counter or anything*", "*...* `while` *has all that counter stuff, I can get lost easier*", "*Because you don't need a counter...*", "*...we have a specific size of* `for`*, we know how many times it will run, so it's much easier to put it in* `for`*, but putting it in a* `while` *and having to worry about updating the counter and in* `for` *it already does this naturally*"



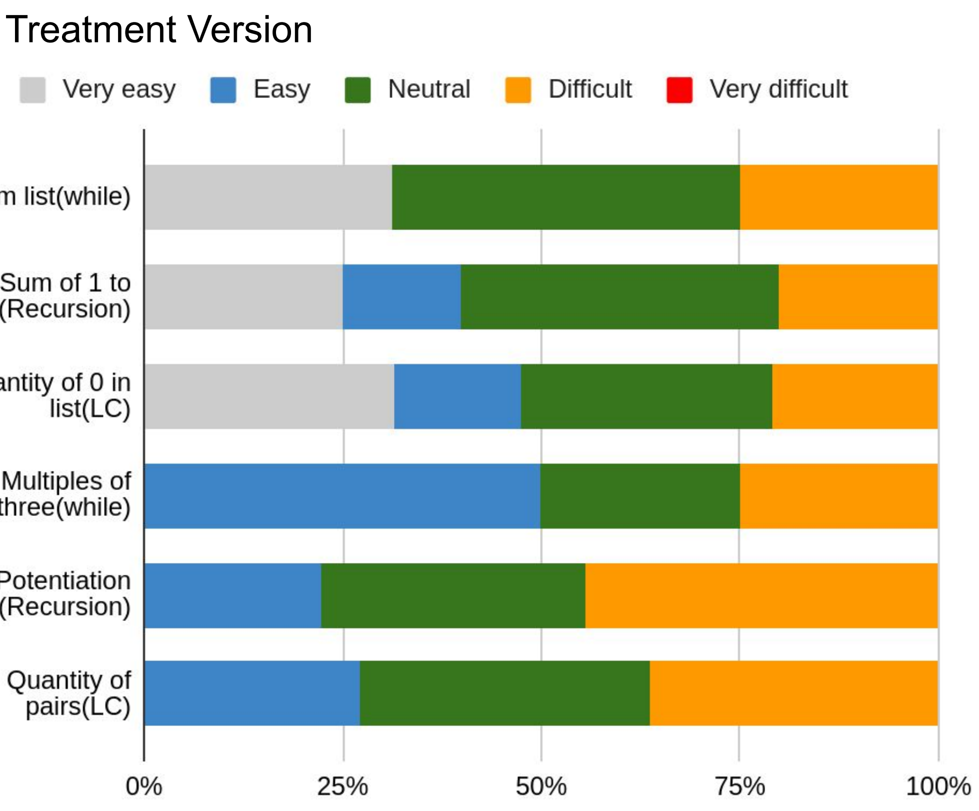
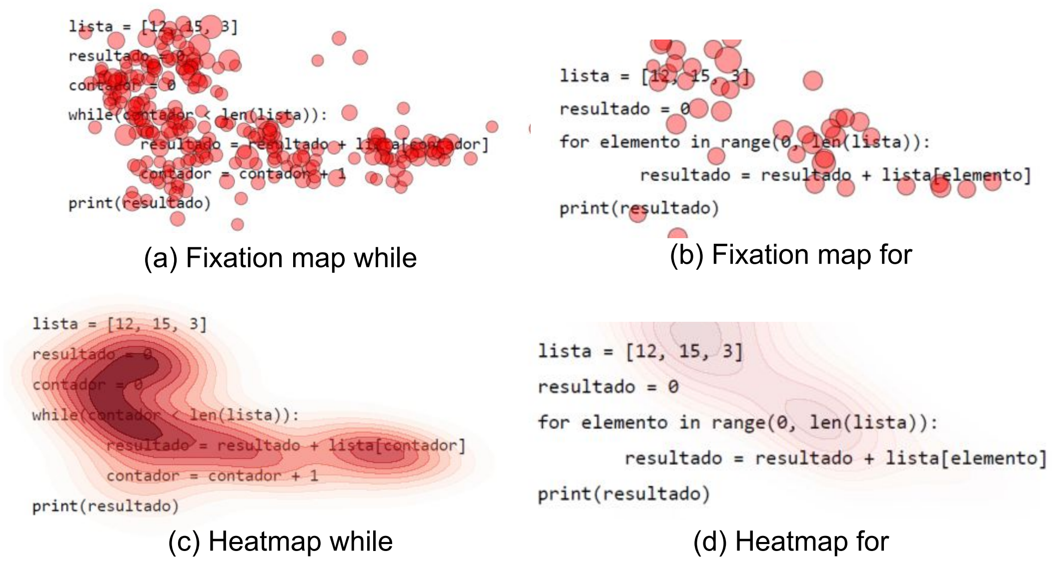


Figure 4.1: Perception of difficulties with the Base and Treatment versions.

As for the `while` version, one of the main reasons given by the subjects for their preference was that it was more explanatory and readable, “`while` *also explains more what it is doing than* `for`*, in this specific* `for`”, “*I find* `while` *easier to read than* `for`”.



```
lista = [12, 15, 3]
resultado = 0
contador = 0
while(contador < len(lista)):
        resultado = resultado + lista[contador]
        contador = contador + 1
print(resultado)
```

```
lista = [12, 15, 3]
resultado = 0
for elemento in range(0, len(lista)):
        resultado = resultado + lista[elemento]
print(resultado)
```

(a) Fixation map while

(b) Fixation map for

(c) Heatmap while

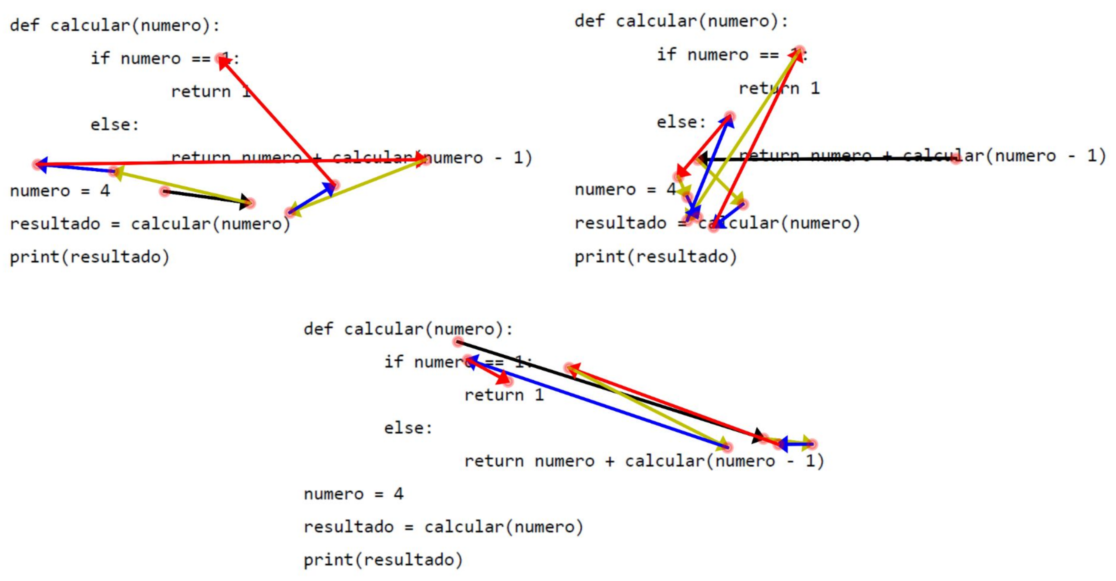
(d) Heatmap for

Figure 4.2: Task *Sum list*, `for` version.

In general, it was possible to observe that tasks with the `for` version compared to the `while` version required less time, number of attempts, number of fixations, duration of fixations and regressions, and some of the main reasons given by the subjects for preferring the `for` version were less variable, it does not use a counter and there is no need to increment the counter, which may indicate an effort in lower overall in terms of understanding. However, the structure with `for` also presents its challenges in Python, such as signs of confusion regarding the use of the `range()` function, as it is necessary to identify where the number of repetitions begins and ends.

4.2.2 Task *Sum from one to n*, Recursion and `for` versions

In Figure 4.3, *Gaze Transitions* from the Recursion version are presented, which are gaze transitions in the task. In this task, the participant used 238.59s, three attempts, 336 fixations, 91.15s in fixation time, 146 regressions and gave up. As mentioned about the difficulty: “*Neutral. It does not have a great degree of complexity, but due to the fact that it calls the* *function within the function itself, the reasoning of thought that has to be carried out to be able to reach it, it is as if it were entering the layer inside the layer and returning, enter inside the layer and return, at least now in a short period of time I couldn't reason to solve it the right way*", it is possible to observe several transitions going back and forth between the base case and the recursive step.



```
def calcular(numero):
        if numero == 1:
                return 1
        else:
                return numero + calcular(numero - 1)
numero = 4
resultado = calcular(numero)
print(resultado)
```

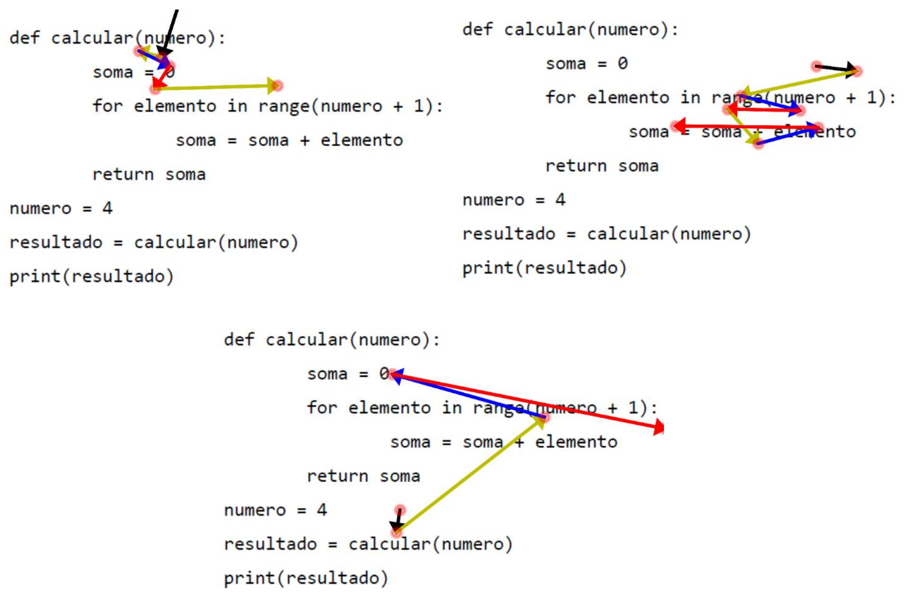
```
def calcular(numero):
        if numero == 1:
                return 1
        else:
                return numero + calcular(numero - 1)
numero = 4
resultado = calcular(numero)
print(resultado)
```

```
def calcular(numero):
        if numero == 1:
                return 1
        else:
                return numero + calcular(numero - 1)
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 4.3: Task *Sum from one to n*, Recursion version.

In Figure 4.4, *Gaze Transitions* of the `for` version are shown. In this task, the participant used 21.05s, one attempt, 24 fixations, 6.18s in fixation time, 9 regressions and solved the task. As mentioned about the main difficulty: "*Understanding what* `for` *was doing. Third line, the declaration of* `for` *would be the most difficult*", it is possible to observe several transitions going back and forth between the structure of `for` and the sum variable.

In general, it was possible to observe that the tasks with the `for` version required less time, fewer attempts and less visual effort than with the Recursion version, which may indicate less effort overall in terms of understanding. This difficulty with recursion may be related to the fact that it is a function that is called repeatedly, requiring dynamic memorization and having a more abstract resolution format. However, it is worth remembering that the structure with `for` also presents its challenges in Python, such as signs of confusion regarding size due to the use of the `range()` function, as it is necessary to add more one to the final value to reach all values.



```
def calcular(numero):
    soma = 0
    for elemento in range(numero + 1):
        soma = soma + elemento
    return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

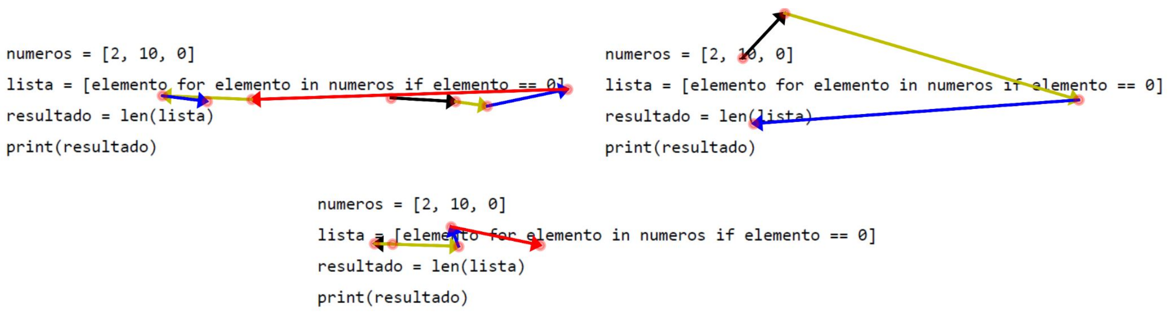
```
def calcular(numero):
    soma = 0
    for elemento in range(numero + 1):
        soma = soma + elemento
    return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

```
def calcular(numero):
    soma = 0
    for elemento in range(numero + 1):
        soma = soma + elemento
    return soma
numero = 4
resultado = calcular(numero)
print(resultado)
```

Figure 4.4: Task *Sum from one to n*, `for` version.

4.2.3 Task *Quantity of 0 in the list*, LC and `for` versions

In Figure 4.5, *Gaze Transitions* of the LC version are shown. In this task, the participant used 25.15s, one attempt, 29 fixations, 8.98s in fixation time, 14 regressions and gave up the task. As mentioned about the main difficulty: "It was this little bit here list = element for element in numbers if element = 2", it is possible to observe several transitions going back and forth between the elements of this line.



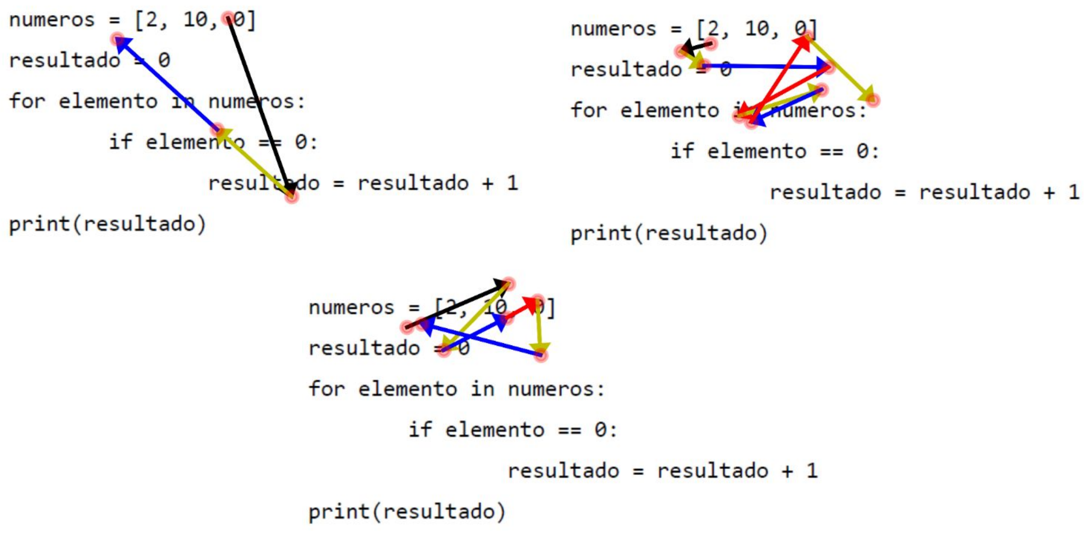
```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

```
numeros = [2, 10, 0]
lista = [elemento for elemento in numeros if elemento == 0]
resultado = len(lista)
print(resultado)
```

Figure 4.5: Task *Quantity of 0 in the list*, LC version.

In Figure 4.6, *Gaze Transitions* of the `for` version are shown. In this task, the participant used 47.47s, one attempt, 41 fixations, 11.77s in fixation time, 18 regressions and solved the task. As mentioned about the strategy used: "*I saw that the result was = 0, and 0 was one of the items in the list of numbers and from what I understood the element would analyze each* *number in the list, the element would become equal to each number of that list, as 0 was an element of that list, then the element inevitably when the last number was counted would be = 0, therefore the result 0 + 1, that is, 1*", it is possible to observe several transitions going back and forth between the elements of that list line.



```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
        if elemento == 0:
                resultado = resultado + 1
print(resultado)
```

```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
        if elemento == 0:
                resultado = resultado + 1
print(resultado)
```

```
numeros = [2, 10, 0]
resultado = 0
for elemento in numeros:
        if elemento == 0:
                resultado = resultado + 1
print(resultado)
```

Figure 4.6: Task *Quantity of 0 in the list*, `for` version.

In general, it was possible to observe that the tasks with the `for` version compared to the LC version required less time, attempts, fixations, duration of fixations and regressions to be solved, which may indicate less effort in the general in terms of understanding. The structure with LC presents its challenges in Python, such as signs of confusion regarding the fact that the definition is longer, and is not well known by beginner developers.

4.2.4 Other Analyses

In this section, we present the distribution of the data obtained. The metrics time, number of attempts to solve the task and visual effort through of fixation duration, fixation count and regressions in relation to the comparisons between the `for`, `while`, Recursion and LC versions.

In Figure 4.7, it is possible to analyze the distribution of data and its density in each comparison between Base (B) which represents the version `for` and Treatment (T) which represents the versions `while`, Recursion and LC. It is possible to observe that although

Comparison	Code				AOI			
Comparison 1	**for (s)**	**while (s)**	**Impact (%)**	**PV**	**for (s)**	**while (s)**	**Impact (%)**	**PV**
	30.54	48.76	↑59.54	0.30	15.77	31.69	↑95.0	0.13
Comparison 2	**for (s)**	**Recursion (s)**	**Impact (%)**	**PV**	**for (s)**	**Recursion (s)**	**Impact (%)**	**PV**
	54.88	62.08	↑13.1	0.31	44.68	50.71	↑13.49	0.57
Comparison 3	**for (s)**	**LC (s)**	**Impact (%)**	**PV**	**for (s)**	**LC (s)**	**Impact (%)**	**PV**
	20.76	32.22	↑55.2	**0.01**	12.72	20.68	↑62.5	**0.02**
All	**for**	**while, LC Recursion**	**Impact (%)**	**PV**	**for**	**while, LC Recursion**	**Impact (%)**	**PV**
	26.88	48.54	↑80.5	**0.01**	16.03	30.91	↑92.7	**0.01**

Table 4.1: Comparative analysis between the code snippets used: Time.

there are outliers and more extreme values, the data density in the Quartiles and the median are similar and often appear a little higher in T compared to B.

4.3 Answers to the Research Questions

Next, we present a summary of the answers to our research questions.

4.3.1 RQ$_1$ What is the impact of code structures on time?

To answer this question, the average time spent on the AOI and on the code as a whole were analyzed, which are presented in Table 4.1. In the AOI region, the `for` version proved to be better than the `while`, Recursion and LC versions. Compared to `for`, overall, increases were observed with the `while`, Recursion and LC versions that ranged from 13.49% and P-Value (PV) 0.57 to 95% and PV 0.13 in the Comparison 2 and Comparison 1 respectively.

As for the complete code, the `for` version proved to be better than the `while`, Recursion and LC versions. Increases were observed with the `while`, Recursion and LC versions that ranged from 13.1% to 59.54% in the Comparison 2 and Comparison 1 respectively.

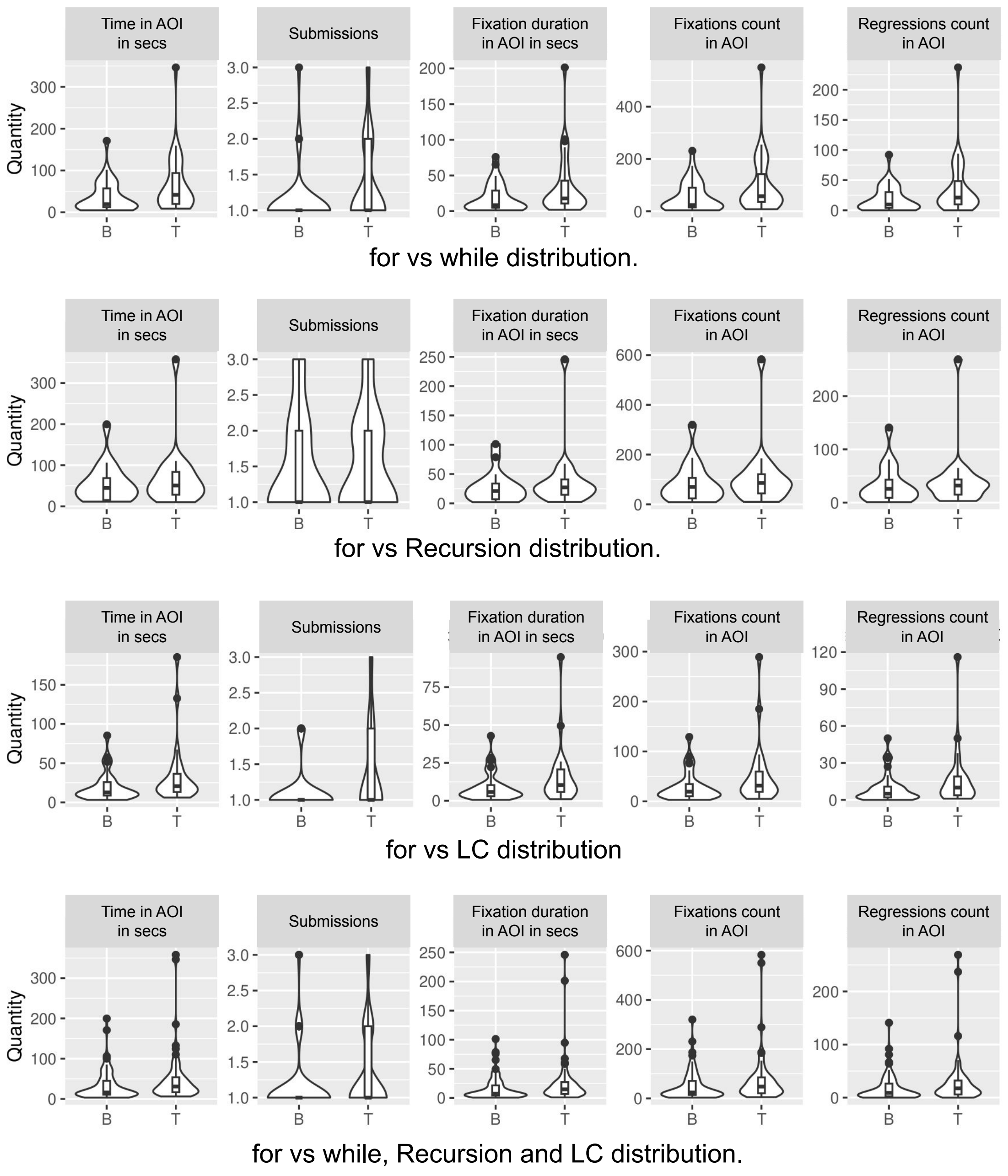


Figure 4.7: for vs while, Recursion and LC distribution.

Comparison	Code			
Comparison 1	**for**	**while**	**Impact (%)**	**PV**
	1	1.12	↑12	0.06
Comparison 2	**for**	**Recursion**	**Impact (%)**	**PV**
	1.47	1.52	↑3.4	0.75
Comparison 3	**for**	**LC**	**Impact (%)**	**PV**
	1.1	1.37	↑24.5	0.06
All	**for**	**while, LC Recursion**	**Impact (%)**	**PV**
	1.14	1.32	↑15.79	0.03

Table 4.2: Comparative analysis between the code snippets used: Number of submissions.

4.3.2 RQ$_2$ What is the impact of code structures on the number of attempts used to solve the task?

To answer this question, the average number of submissions were analyzed, which are presented in Table 4.2. The `for` version was better than the `while`, Recursion and LC versions. Compared to `for`, overall, increases were observed with the `while`, Recursion and LC versions that ranged from 3.4% and PV 0.75 to 24.5% and PV 0.06 in the Comparison 2 and Comparison 3 respectively.

4.3.3 RQ$_3$ What is the impact of code structures on the duration of fixes?

To answer this question, the average duration of fixations in the full code and in the AOI were analyzed, which are presented in Table 4.3. In the AOI region, the `for` version was better than the `while`, Recursion and LC versions. Compared to `for`, overall, increases were observed with the `while`, Recursion and LC versions that ranged from 29.76% and PV 0.43 to 97.22% and PV 0.13 in the Comparison 2 and Comparison 1, respectively.

As for the complete code, the `for` version was better than the `while`, Recursion and LC versions in three tasks, but it was worse than `while` and LC in two tasks. Overall, increases

Comparison	Code				AOI			
Comparison 1	**for (s)**	**while (s)**	**Impact (%)**	**PV**	**for (s)**	**while (s)**	**Impact (%)**	**PV**
	13.87	20.39	↑47.02	0.19	7.20	14.2	↑97.22	0.13
Comparison 2	**for (s)**	**Recursion (s)**	**Impact (%)**	**PV**	**for (s)**	**Recursion (s)**	**Impact (%)**	**PV**
	27	31.31	↑15.93	0.33	21.06	27.33	↑29.76	0.43
Comparison 3	**for (s)**	**LC (s)**	**Impact (%)**	**PV**	**for (s)**	**LC (s)**	**Impact (%)**	**PV**
	8.98	13.57	↑51.00	0.06	5.74	10.39	↑80.96	0.06
All	**for**	**while, LC Recursion**	**Impact (%)**	**PV**	**for**	**while, LC Recursion**	**Impact (%)**	**PV**
	12.59	20.9	↑65.92	**0.02**	7.31	14.79	↑232.87	**0.02**

Table 4.3: Comparative analysis between the code snippets used: Duration of fixations.

were observed with the `while`, Recursion and LC versions that ranged from 15.93% to 51% in the comparison 2 and Comparison 3, respectively.

4.3.4 RQ$_4$ What is the impact of code structures on the number of fixations?

To answer this question, the average number of fixations in the complete code and in the AOI were analyzed, which are presented in Table 4.4. In the AOI region, the `for` version was better than the `while`, Recursion and LC versions. Compared to `for`, overall, increases were observed with the `while`, Recursion and LC versions that ranged from 22.86% and PV 0.34 to 95.45% and PV 0.12 in the Comparison 2 and Comparison 1, respectively.

As for the complete code, the `for` version was better than the `while`, Recursion and LC versions in three tasks, but it was worse than `while` and LC in two tasks. Overall, increases were observed with the `while`, Recursion and LC versions that ranged from 25.97% to 59.49% in the Comparison 2 and Comparison 1, respectively.

Comparison	Code				AOI			
Comparison 1	**for**	**while**	**Impact (%)**	**PV**	**for**	**while**	**Impact (%)**	**PV**
	39.5	63	↑59.49	0.21	22	43	↑95.45	0.12
Comparison 2	**for**	**Recursion**	**Impact (%)**	**PV**	**for**	**Recursion**	**Impact (%)**	**PV**
	77	97	↑25.97	0.28	70	86	↑22.86	0.34
Comparison 3	**for**	**LC**	**Impact (%)**	**PV**	**for**	**LC**	**Impact (%)**	**PV**
	28	44	↑57.14	0.06	19.5	31.5	↑61.54	0.08
All	**for**	**while, LC Recursion**	**Impact (%)**	**PV**	**for**	**while, LC Recursion**	**Impact (%)**	**PV**
	38	63	↑65.79	**0.02**	24	48	↑100	**0.02**

Table 4.4: Comparative analysis between the code snippets used: Fixations.

4.3.5 RQ$_5$ What is the impact of code structures on the number of regressions?

To answer this question, the average number of regressions in the complete code and in the AOI were analyzed, which are presented in Table 4.5. In the AOI region, the `for` version was better than the `while`, Recursion and LC versions. Compared to `for`, in general, increases were observed with the `while`, Recursion and LC versions that ranged from 23.08% and PV 0.35 to 114.29% and PV 0.15 in the Comparison 2 and Comparison 1, respectively.

As for the complete code, the `for` version was better than the `while`, Recursion and LC versions. Overall, increases were observed with the `while`, Recursion and LC versions that ranged from 48.57% to 90% in the Comparison 1 and Comparison 3, respectively.

4.4 Threats to Validity

In this section, threats to internal validity will be discussed in Section 4.4.1, external validity in Section 4.4.2, and construct validity in Section 4.4.3.

Comparison	**Code**				**AOI**			
Comparison 1	**for**	**while**	**Impact (%)**	**PV**	**for**	**while**	**Impact (%)**	**PV**
	17.5	26	↑48.57	0.19	7	15	↑114.29	0.15
Comparison 2	**for**	**Recursion**	**Impact (%)**	**PV**	**for**	**Recursion**	**Impact (%)**	**PV**
	30	45	↑50	0.21	26	32	↑23.08	0.35
Comparison 3	**for**	**LC**	**Impact (%)**	**PV**	**for**	**LC**	**Impact (%)**	**PV**
	10	19	↑90	0.09	5	10	↑100	0.10
All	**for**	**while, LC Recursion**	**Impact (%)**	**PV**	**for**	**while, LC Recursion**	**Impact (%)**	**PV**
	15	26.5	↑76.67	**0.02**	9	17.5	↑94.44	**0.03**

Table 4.5: Comparative analysis between the code snippets used: Regressions.

4.4.1 Internal Validity

The experiment was conducted in different locations, which may have influenced the participants' visual attention. For mitigation, the locations were organized to have similar conditions of light, comfort, temperature and stillness.

The presence of a researcher on site may affect participants' visual attention or performance. For mitigation, we sought to avoid interactions during each task.

Eye tracking equipment has limitations, even with calibration and recalibrations, adjustments to the captured points were necessary. For some participants, the heat map and fixations showed a red color and fixations in white areas, places with no code. For these cases, a small adjustment was necessary. All fixations for a given program received the same adjustment. The adjustments to the y-coordinate were between 10 and 17, the x-coordinate was not adjusted. This adjustment of points can influence the interpretation; these adjustments were discussed by the researchers. However, the threat of adjusting the points would be preferable to the threat of analyzing the data with points that were not touching the code.

A fixed chair was used in the experiment, to avoid compromising data collection by the eye tracking camera. For each participant, approximately one hour was allocated, and six tasks were assigned to each participant and at the end an interview. Which may have influenced visual effort. Therefore, simple and short programs were designed.

The result may have been influenced by the order in which students learned the concepts. Aiming to mitigate this potential bias, the experiment was conducted in three different educational institutions, located in two different cities. Furthermore, the Latin Square was implemented.

4.4.2 External Validity

Short programs were used to make it possible to view the entire screen. Which may restrict generalization to larger programs. The study focused on Python novices, which may restrict generalization to more experienced Python developers. Other eye-tracking studies have also focused on novices to understand code comprehension.

The study focused on Python, which may restrict generalization to other programming languages. For mitigation, common constructions in other languages were used. The programs were designed with indicators in Portuguese, as the participants were Brazilian.

The answers to the elaborated tasks were numerical values and had to respond to the program output aloud. This task may not generalize to other tasks, such as adding a resource for example.

The names of methods and variables can influence participants' understanding and visual effort. Confusing names can make it difficult, just as certain names can make it significantly easier. For mitigation, the names were refined and discussed among researchers.

4.4.3 Construction Validity

Code understanding has often been measured through response time and correctness. Time, response correction, and visual effort were also combined to investigate code comprehension.

When inviting participants, you need to make them aware that their eyes are being tracked. Which can influence where or how much they look at certain locations on the screen. To minimize this threat, the precise objectives of the study are not stated, avoiding assumptions.

Chapter 5

Related Work

In this section, related works will be presented. In Section 5.1, works that focus on comparing different structures and investigating participants' understanding will be presented. In Section 5.2, some approaches to measuring code comprehension in this context will be presented.

5.1 Code Structure Comparison

Endres et al. [21] conducted a study with 162 undergraduate students comparing different iterative, recursive, and tail-recursive code structures through task correctness. First, the authors performed a task-specific analysis and then investigated differences in students' most common errors by program structure. They found that students were more likely to produce wrong answers with incorrect types or structures for recursive and tail recursive versions of programs. They investigated correlations between programming performance and other factors such as experience, gender, ethnicity and others, among which programming experience proved to be the most significant factor. Similarly, in the present study, different code structures were also investigated in the context of novices with respect to time spent, comments and the number of attempts. Furthermore, they were triangulated with visual effort, which made it possible to identify significant relationships between these factors and the visual effort expended by the participants.

Sulov [68] empirically compared and investigated the preference of novice students with respect to the Iteration and Recursion code structures. The study investigated the resolution of tasks in the C language with 130 students undergoing an introduction to programming. The authors investigated students' preference and success rate when dealing with programming tasks, which could be solved with iteration or recursion. The results showed that programming novices prefer Iteration to Recursion in most cases. The results also showed that as students gain more experience and move from abstract theory to real software, they do not always identify possible cases in which Recursion should be preferred. Our work investigated 32 students in introductory programming with respect to preference, success rate, task solving time and comprehension while the novices dealt with programming tasks with Iteration, Recursion and List Comprehension. The language used was Python. It was noticed that little experience can cause a discrepancy between the declared preference and the performance presented in the tasks.

Turbak et al. [69] describe an experience report, they investigate the teaching of Recursion prior to the teaching of loops and Iteration. The study measured the number of visits during office hours and student performance on problem sets and exams in the course. They argue that there are strong theoretical, practical, and pedagogical reasons to teach Recursion before loops in a Computer Science course, regardless of the programming paradigm that is taught. Some arguments are raised in the work, among them are: Recursion is more fundamental than the loop, Recursion requires fewer prerequisites than loop and deep ideas take time to be absorbed. They observed that students seemed to leave their course with better problem-solving skills than they had before. They believe this is largely due to the fact that they now place more emphasis on the divide, conquer, and paste strategy. In our work, we also investigated how programming novices solve problems with Recursion and Iteration. By measuring the number of attempts, it was noticed that Recursion in general required more attempts than Iteration (`for`) and obtained a greater number of dropouts, while the time spent was similar, but worse for Recursion.

Haberman and Averbuch [32] conducted a study focusing on students' difficulties with Recursion in relation to the base cases. Some findings of the study revealed that students face difficulties in identifying problem-based cases. They tend to deal with redundant base cases, ignore boundary values and degenerate cases, avoid considering values outside the proper range, and in some cases even fail to define any base cases when designing recursive algorithms. Furthermore, students encounter difficulties when evaluating recursive algorithms that deal with base cases that are difficult to identify initially. Similarly, in our work, base cases were also investigated, including through eye tracking, which made it possible to identify that the base case is one of the locations in the Recursion code most viewed by participants.

Esteero et al. [22] investigated what CS2 students chose when asked to solve a problem that could be solved with Recursion or Iteration and how this choice relates to the correctness of their code. They sought to provide an answer to this question through an analysis of students' exam answers to a problem about finding the deepest common ancestors in trees. They found that 19% of students chose to use Iteration, 51% chose Recursion, and 16% chose the combination of Iteration and Recursion. Regarding correctness, students who chose Iteration performed better than those who chose Recursion and the combination of both. Similarly, in our work, participants' preferences and their respective performance on the tasks were investigated using the number of attempted responses in addition to time and visual effort. It was identified that, in the context of novices, there are signs of disagreement between preference and real performance.

Mccauley et al. [39] examined the computing education research literature, presenting their findings on the challenges students face in learning Recursion, the mental models students develop as they learn Recursion. We revisited more than 35 publications documenting research results related to teaching and learning recursive programming and many of these studies compare the effectiveness of introducing Iteration before Recursion and vice versa. In our work, eye tracking was used, which allowed a thorough analysis from the point of view of visual effort, in addition, List Comprehension was introduced into the comparison.

5.2 Approaches to Measuring Code Comprehension

Researchers have been looking for new ways to investigate code understanding using more objective metrics, going beyond comparing preferences. For example, Da Costa et al. [17] compared two code structures with #ifdef annotations, one with disciplined annotations, and one with undisciplined annotations. They assessed whether disciplining #ifdef annotations correlates with improvements in code comprehension and eye strain using an eye tracker. A controlled experiment was conducted with 64 individuals, most of whom were new to the C programming language. Statistically significant differences were observed in relation to the analyzed metrics (time, duration of fixations, number of fixations and number of regressions) in the code regions changed by each refactoring.

Another study conducted by Da Costa et al. [16] conducted a controlled experiment comparing two code structures, one with the presence of small clutter patterns called atoms and the other without the presence of these patterns. The study included 32 Python novices and measured their time, number of attempts, and visual effort using an eye tracker. The authors also conducted interviews and investigated participants' difficulties with the programs. In our work, a similar design was used in terms of metrics and use of eye tracking, however, in a different context, comparing Recursion, Iteration and List Comprehension structures.

De Oliveira et al. [19] conducted an experiment with 30 students and software professionals, using eye tracking to assess whether developers understand code with the presence of confusion atoms. To do this, task completion time and response accuracy were measured and the distribution of visual attention was analyzed. The authors compared code snippets from real open source C/C++ systems from different domains containing three types of atoms. Their findings reinforce that atoms hinder developers' performance and understanding. In our work, a similar design was used in terms of metrics and use of eye tracking, however, in a different context and exploring fixation duration, fixation count and regression count, comparing Recursion, Iteration and List Comprehension structures. in the Python language.

Sharif and Maletic [60] presented an empirical study to determine whether identifier naming conventions (camelCase and under_score) affect code comprehension. Time, correctness of responses, and visual effort were measured using eye tracking. They found a significant improvement in time and visual effort with the underscore style. Another study conducted by Sharafi et al. [59] reported the results of an experiment involving 15 male and nine female participants to study the impact of gender on participants' visual effort, time, and recall accuracy. camelCase and underscore identifiers when reading the code. No statistically significant differences were observed regarding metrics. In our work, in addition to these metrics, the number of submissions was also measured.

Siegmund et al. [62] conducted work in which they explored whether functional magnetic resonance imaging (fMRI), which is well established in cognitive neuroscience, would be viable to more directly measure program understanding. To do this, the researchers carried out a controlled experiment, in which 17 participants were observed inside an fMRI scanner, while the participants understood small snippets of code, which were compared with the location of syntax errors. They found a clear and distinct activation pattern of five brain regions related to working memory, attention and language processing, which are processes that fit well with program comprehension knowledge. In our work, an eye tracking camera was used; this tool has also demonstrated great potential for analyzing code comprehension. A controlled study was conducted, using a camera attached to the notebook while participants solved small snippets of code. The data generated by the camera made it possible to project graphics such as heat maps, fixations and paths taken by the eyes, which allowed an analysis of code understanding from the point of view of visual effort.

Chapter 6

Conclusions

In this work, a controlled experiment with eye tracking was carried out to evaluate the impact of Recursion, Iteration and LC on the code comprehension of Python novices. The versions were compared across six tasks measuring the impact on time, number of attempts, and visual effort of 32 participants. Performance metrics were triangulated with semi-structured interviews carried out with participants and the results of their preferences.

Regarding the impact of the Recursion, Iteration and LC structures on the time spent in the AOI, compared to `for`, in general, an increase was observed with the `while`, Recursion and LC versions that ranged from 13.49% up to 95% in Recursion and `while`, respectively. Therefore, for the tasks evaluated, with respect to time, the `for` version proved to be preferable. Regarding the number of attempts, compared to `for`, an increase of 24.5% was observed in the number of submissions with the LC version, showing evidence that this version is associated with greater confusion among participants.

With regard to visual effort, an impact was also observed. In the AOI region, regarding the duration of fixations, compared to `for`, in general, increases were observed with the `while`, Recursion and LC versions that varied from 29.76% to 97.22% with Recursion and `while`, respectively, indicating that there was a need to look longer. Regarding the number of fixations, compared to `for`, in general, increases were observed with the `while`, Recursion and LC versions that varied from 22.86% to 95.45% with Recursion and `while`, respectively, indicating that there was a need to focus on more places. Regarding the number of regressions, compared with `for`, in general, increases were observed with the `while`, Recursion and LC versions that varied from 23.08% to 114.29% with Recursion and `while`, respectively, indicating the need to return more times in the code. Therefore, in general, the `for` version was associated with less visual effort on the part of novices for the tasks evaluated.

The analysis of code reading patterns also presented important clues. Through the analysis of reading patterns, in Recursion, greater focus of attention was noticed in the base case and in the recursive step. The need for a stop condition and explicit counter is a hypothesis for the worst performance with `while`. There were agreements and discrepancies among participants between performance and perception of difficulty depending on the task. In general, the `for` version required less time in the AOI along with less visual effort, indicating a better understanding of some tasks.

This study contributes to educating educators about the impact of Recursion, Iteration and LC on the code understanding of Python novices. These implications can impact the way of teaching, promoting greater code understanding by novices with less visual effort. Furthermore, it raises the awareness of researchers to use eye tracking as a research tool that shows nuances that metrics based solely on code analysis may not be able to capture.

6.1 Future work

As the next steps of the research, we intend to evaluate Python constructions in more tasks, for example, other arithmetic operations, tasks that use more conditional statements, string manipulations, file manipulations, among others. The application of the structures in other types of tasks and contexts will contribute to obtaining other perspectives and other insights regarding the use of Python constructions.

Since there are several Python constructions, we can evaluate other constructions such as conditional statements (if, else, elif), List, Tuple, Dictionary, Exception Treatment (try, except, finally), classes and objects, among others. We intend to carry out a controlled experiment with a larger number of Python beginners, which will contribute to obtaining tests and statistical differences in addition to other insights based on more qualitative data.

Since there are similarities between programming languages, we can also evaluate structures that present similarities in different programming languages such as Java, JavaScript, Swift, among others. Varying the language can impact the size of the code snippets. We also aim to explore larger code snippets with more advanced tools that allow scrolling the content such as iTrace [31], in addition to tools that support source code edits in eye-tracking studies such as iTrace-Atom [23]. Varying the language and size of code snippets can give us a better understanding of the impact of certain code constructions on code comprehension.

For a better qualitative analysis, we can carry out interviews with students, as well as interviews with teachers. We can triangulate the interviews with the data obtained through quantitative metrics. This will allow a better understanding of the reading patterns used to solve the tasks, in addition to a better understanding of the reasons why the subjects did or did not understand the structures. In addition, we can employ rigorous qualitative method analysis such as grounded theory as proposed by Strauss and Corbin [67]. Grounded theory allows one to understand a phenomenon, such as code comprehension, however, without preconceived theories. The theory has to emerge fro the data.

Another way to improve the analysis of quantitative data is to triangulate it with other forms of analysis. We can use other types of equipment that make it possible to obtain other types of data, for example, the neural part, heartbeats using smartwatches and smartbands while the subjects solve the tasks. This will allow other analyses, through data triangulation. Such combinations of psycho-physiological measures has been employed before to assess task difficulty in software development [26].

We can also assist in the development of more modern tools that have IDEs coupled with eye tracking cameras and can identify patterns that are difficult to understand and automatically modify the code snippet. In this way, novices would be helped in an automatic way, enabling a better understanding.

We can also evaluate the same study with professionals, and thus, compare the results found with novices, in terms of visual metrics, time, submissions and answers given in interviews. This will allow us to identify and analyze the similarities and differences between the reading patterns used by professionals and novices.

We are also planning to extend our evaluation to code snippets from various domains utilizing eye-tracking technology. Specifically, our objective is to investigate whether code [46; 45; 44; 48; 37; 5; 6], configurable systems [41; 42; 38; 43], testing [65; 64], model [27; 28; 29] refactorings, and the application of quick fixes [20; 53] improve quality. This approach will allow us to assess the impact of these practices on the overall quality of software development processes and outcomes.

Bibliography

[1] Nahla J. Abid, Jonathan I. Maletic, and Bonita Sharif. Using developer eye movements to externalize the mental model used in code summarization tasks. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research and Applications*, ETRA '19. ACM, June 2019.

[2] Duane A Bailey. Python structures. 2013.

[3] Victor Basili, G. Caldiera, and H. Rombach. The Goal Question Metric Approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[4] Roman Bednarik and Markku Tukiainen. An Eye-tracking Methodology for Characterizing Program Comprehension Processes. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, ETRA'06, pages 125–132, 2006.

[5] Ana Carla Bibiano, Wesley K. G. Assunção, Daniel Coutinho, Kleber Santos, Vinícius Soares, Rohit Gheyi, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Daniel Oliveira, Caio Barbosa, João Lucas Marques, and Anderson Oliveira. Look ahead! revealing complete composite refactorings and their smelliness effects. In *International Conference on Software Maintenance and Evolution*, pages 298–308. IEEE, 2021.

[6] Ana Carla Bibiano, Vinícius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. How does incomplete composite refactoring affect internal quality attributes? In *International Conference on Program Comprehension*, pages 149–159. ACM, 2020.

[7] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and

Bonita Sharif. The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[8] Tanja Blascheck and Bonita Sharif. Visually analyzing eye movements on natural language texts and source code snippets. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research and Applications*, ETRA '19, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Agnieszka Aga Bojko. Informative or misleading? heatmaps deconstructed. In *International conference on human-computer interaction*, pages 30–39. Springer, 2009.

[10] George EP Box, J Stuart Hunter, and William G Hunter. Statistics for experimenters. In *Wiley series in probability and statistics*. Wiley Hoboken, NJ, 2005.

[11] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.

[12] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. 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, pages 1–9, 2011.

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

[14] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *PPIG*, page 5, 2002.

[15] Martha E Crosby and Jan Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):25–35, 1990.

[16] José Aldo Silva da Costa, Rohit Gheyi, Fernando Castor, Pablo Roberto Fernandes de Oliveira, Márcio Ribeiro, and Baldoino Fonseca. Seeing confusion through a new

lens: on the impact of atoms of confusion on novices' code comprehension. *Empirical Software Engineering*, 28(4):81, 2023.

[17] José Aldo Silva da Costa, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Vander Alves, Baldoino Fonseca, Flávio Medeiros, and Alessandro Garcia. Evaluating refactorings for disciplining# ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, 26(5):92, 2021.

[18] Daniel Kyle Davis and Feng Zhu. Analysis of software developers' coding behavior: A survey of visualization analysis techniques using eye trackers. *Computers in Human Behavior Reports*, 7:100213, 2022.

[19] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. Atoms of confusion: The eyes do not lie. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 243–252, 2020.

[20] Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories. In *Brazilian Symposium on Software Engineering*, pages 74–83. ACM, 2021.

[21] Madeline Endres, Westley Weimer, and Amir Kamil. An analysis of iterative and recursive problem performance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 321–327, 2021.

[22] Ramy Esteero, Mohammed Khan, Mohamed Mohamed, Larry Yueli Zhang, and Daniel Zingaro. Recursion or iteration: Does it matter what students choose? In *Proceedings of the 49th ACM technical symposium on computer science education*, pages 1011–1016, 2018.

[23] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan Maletic. gazel: supporting source code edits in eye-tracking studies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 69–72. IEEE, 2021.

[24] Dror G Feitelson. Considerations and pitfalls in controlled experiments on code comprehension. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 106–117. IEEE, 2021.

[25] Dror G Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software Engineering*, 27(6):123, 2022.

[26] Thomas Fritz, Andrew Begel, Sebastian Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the International Conference on Software Engineering*, ICSE'14, pages 402–413, 2014.

[27] Rohit Gheyi and Paulo Borba. Refactoring alloy specifications. *Elsevier's Electronic Notes in Theoretical Computer Science*, 95:227–243, 2004.

[28] Rohit Gheyi, Tiago Massoni, and Paulo Borba. An abstract equivalence notion for object models. *Elsevier's Electronic Notes in Theoretical Computer Science, Proceedings of Brazilian Symposium on Formal Methods 2004*, 130:3–21, 2005.

[29] Rohit Gheyi, Tiago Massoni, and Paulo Borba. Automatically checking feature model refactorings. *Journal of Universal Computer Science (JUCS)*, 17:684–711, 2011.

[30] Drew Guarnera, Corey Bryant, Ashwin Mishra, Jonathan Maletic, and Bonita Sharif. iTrace: Eye tracking infrastructure for development environments. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, ETRA'18. ACM, 2018.

[31] Drew T Guarnera, Corey A Bryant, Ashwin Mishra, Jonathan I Maletic, and Bonita Sharif. itrace: Eye tracking infrastructure for development environments. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, pages 1–3, 2018.

[32] Bruria Haberman and Haim Averbuch. The case of base cases: Why are they so difficult to recognize? student difficulties with recursion. In *Proceedings of the 7th annual conference on innovation and technology in computer science education*, pages 84–88, 2002.

[33] Alexander Homann, Lisa Grabinger, Florian Hauser, and Jürgen Mottok. An eye tracking study on misra c coding guidelines. In *Proceedings of the 5th European Conference on Software Engineering Education*, ECSEE 2023. ACM, June 2023.

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

[35] Marcel A Just and Patricia A Carpenter. A Theory of Reading: From Eye Fixations to Comprehension. *Psychological review*, 87(4):329, 1980.

[36] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education*, 22(2):1–33, November 2021.

[37] Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, Rohit Gheyi, Márcio Ribeiro, Alessandro F. Garcia, and Rafael Maiani de Mello. Understanding and detecting harmful code. In *Brazilian Symposium on Software Engineering*, pages 223–232. ACM, 2020.

[38] Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. The discipline of preprocessor-based annotations does #ifdef TAG n't #endif matter. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 297–307, 2017.

[39] Renée McCauley, Scott Grissom, Sue Fitzgerald, and Laurie Murphy. Teaching and learning recursive programming: a review of the research literature. *Computer Science Education*, 25(1):37–66, 2015.

[40] Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. Investigating misunderstanding code patterns in C open-source software projects. *Empirical Software Engineering*, 24:1693–1726, 2019.

[41] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of

preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2018.

[42] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Larissa Braz, Christian Kästner, Sven Apel, and Kleber Santos. An empirical study on configuration-related code weaknesses. In *Brazilian Symposium on Software Engineering*, pages 193–202. ACM, 2020.

[43] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015*, pages 35–44, 2015.

[44] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44(5):429–452, 2018.

[45] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39–64, 2014.

[46] Melina Mongiovi, Gustavo Wagner, Rohit Gheyi, Gustavo Soares, and Marcio Ribeiro. Scaling testing of refactoring tools. In *International Conference on Software Maintenance and Evolution*, 2014.

[47] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–359. IEEE, 2020.

[48] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. Revisiting the refactoring mechanics. *Information & Software Technology*, 110:136–138, 2019.

[49] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM*

43rd International Conference on Software Engineering (ICSE), pages 524–536. IEEE, 2021.

[50] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C Hofmeister, and André Brechmann. Simultaneous measurement of program comprehension with fmri and eye tracking: A case study. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.

[51] Keith Rayner. Eye movements in reading and information processing. *Psychological bulletin*, 85(3):618, 1978.

[52] Rachana S Rele and Andrew T Duchowski. Using eye tracking to evaluate alternative search results interfaces. In *Proceedings of the human factors and ergonomics society annual meeting*, volume 49, pages 1459–1463. SAGE Publications Sage CA: Los Angeles, CA, 2005.

[53] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 404–415, 2017.

[54] Jonathan A. Saddler, Cole S. Peterson, Sanjana Sama, Shruthi Nagaraj, Olga Baysal, Latifa Guerrouj, and Bonita Sharif. Studying developer reading behavior on stack overflow during api summarization tasks. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 195–205, 2020.

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

[56] Timothy R Shaffer, Jenna L Wise, Braden M Walters, Sebastian C Müller, Michael Falcone, and Bonita Sharif. itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 954–957, 2015.

[57] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. Eye-tracking metrics in software engineering. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 96–103. IEEE, 2015.

[58] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. A Practical Guide on Conducting Eye Tracking Studies in Software Engineering. *Empirical Software Engineering*, 25(5):3128–3174, 2020.

[59] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Women and men—different but equal: On the impact of identifier style on source code reading. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 27–36. IEEE, 2012.

[60] Bonita Sharif and Jonathan I Maletic. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 196–205. IEEE, 2010.

[61] Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, March 2016.

[62] Janet Siegmund, Christian Kástner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th international conference on software engineering*, pages 378–389, 2014.

[63] José Aldo Silva Da Costa and Rohit Gheyi. Evaluating the code comprehension of novices with eye tracking. In *Proceedings of the XXII Brazilian Symposium on Software Quality*, pages 332–341, 2023.

[64] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André L. M. Santos. Refactoring test smells: A perspective from open-source developers. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 50–59. ACM, 2020.

[65] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. Refactoring test smells with JUnit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 2022.

[66] Thierry Sorg, Amine Abbad-Andaloussi, and Barbara Weber. Towards a fine-grained analysis of cognitive load during program comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, March 2022.

[67] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research Techniques*. Thousand Oaks, CA: Sage publications, 1998.

[68] Vladimir Sulov. Iteration vs recursion in introduction to programming classes: an empirical study. *Cybernetics and Information Technologies*, 16(4):63–72, 2016.

[69] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before loops in cs1. *Journal of Computing in Small Colleges*, 14(4):86–101, 1999.

[70] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 231–234, 2014.

[71] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8–style guide for python code. *Python. org*, 1565:28, 2001.

[72] Braden Walters, Michael Falcone, Alexander Shibble, and Bonita Sharif. Towards an eye-tracking enabled ide for software traceability tasks. In *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 51–54. IEEE, 2013.

[73] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.