eyeCode: An Eye-Tracking Experimental Framework for Program Comprehension

Michael Hansen (mihansen@indiana.edu**)**
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Andrew Lumsdaine (lums@indiana.edu**)**
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Robert L. Goldstone (rgoldsto@indiana.edu**)**
Dept. of Psychological and Brain Sciences, 1101 E. 10th Street
Bloomington, IN 47405 USA

October 10, 2014

Abstract

Psychologists and computer scientists have studied the cognitive aspects of programming for nearly thirty years. Eye-tracking has gained popularity recently as a tool for gaining insight into programmer's cognitive processes while they comprehend programs. Both low-level eye movement metrics, such as fixation duration, and high-level metrics like line to line transition probabilities, provide rich alternatives to standard think-aloud experiment protocols. We present an experiment in which programmers predict the output of ten short Python programs, each with alternative versions. We use eye movements to discover important code elements/lines, characterize evaluation strategies, and investigate the source of participant errors. In addition, an open source data analysis library is introduced that provides specialized metrics, plots, and statistics for eye-tracking program comprehension experiments.

1 Introduction

The cognitive aspects of programming have been studied by psychologists and computer scientists for nearly thirty years [15], culminating in the development of several **cognitive models** of program comprehension (e.g., [37, 31, 9, 16]). Though useful as tools to describe and reason about program comprehension, these (mostly qualitative) models cannot *quantitatively* predict human behavior when comprehending a specific program. The development of a quantitative cognitive model would represent a milestone in the understanding of program comprehension, and facilitate the semi-automated analysis of design alternatives for programming languages and integrated development environments (IDEs).

Recently, the use of **eye-tracking** has become prevalent in the study of program comprehension [38, 1, 8]. Eye movements are a rich data source, and have been strongly linked with visual attention and cognitive processes [29]. By collecting and analyzing eye movement data from developers during a specific programming task, researchers hope to gain insight into the cognitive processes programmers use to read and understand programs. When combined with other sources of data, such as responses to questionnaires, task timing, response accuracy, and participant experience, the space of possible cognitive models can be strongly constrained. Our programming task, predicting a program's printed output, is a starting point for the development of a cognitive model. This model will predict human behavior when interpreting simple Python programs.

1.1 The Experiment and Research Questions

We present an experiment in which 29 participants with a range of Python and overall programming experience predicted the output of ten small Python programs. Most of the program texts were less than twenty lines long and performed simple calculations (e.g., computed the area of a rectangle). We used ten different program **bases**, each of which had two or three **versions** with subtle differences. Participants were randomly assigned a version of each program, and performed the experiment in front of a Tobii TX300 free-standing eye-tracker (recording at 300Hz). We computed a variety of eye movement metrics, and produced high-level static and dynamic summary statistics of our data. These metrics and statistics, such as time spent looking at particular code elements and transition probabilities between code lines, helped us answer several research questions.

First, **how does the eye movement data from our experiment compare to other eye-tracking program comprehension experiments?** Our task, output prediction, is relatively unique compared to other studies. Locating bugs and answering comprehension questions are the most common tasks (see [8] for an brief survey). Like many other eye-tracking studies, we make use of common fixation metrics and relationships between areas of interest (AOIs) to summarize our data. In addition, we incorporate specific program comprehension measures – e.g., proportion of lines reviewed in the first 30% of a trial [38] – and compute fixation metrics over rolling time windows.

Second, **can aggregate eye movement metrics and summary statistics be predicted from textual/syntactic features of code?** Readability studies of code have found that both textual features (whitespace, word/line length), and syntactic features of code (identifier/keyword count) can influence assessments of the perceived complexity of the code and reading behavior [6, 13]. By linking code features with eye movements, it may be possible to predict how difficult a specific program is to read and comprehend.

Another benefit of these studies is the contribution of valuable empirical data towards the investigation of programming language usability [35]. Language usability studies are rare, and the addition of new language features seldom involves controlled human experiments. A quantitative understanding of program comprehension would be a boon to researchers and members of industry who are examining language design alternatives.

Finally, **do differences between versions of the same program, or demographics/performance of the participant, influence eye movements?** Eye movement differences have been previously observed between expert and novice programmers during comprehension and debugging tasks [2, 4]. While more experienced programmers tend to perform better, researchers have noted differences in *strategies* between more and less experienced developers. In our experiment, we consider both a participant's Python and overall programming expertise, as well as which *version of a program* they were asked to interpret. Some program versions were designed to reward syntactic and semantic knowledge of Python (e.g., `counting`, `scope`), while others were intended to expose performance differences (e.g., `rectangle`, `whitespace`). When analyzing our results, we separated trials for some programs by response correctness, and compared eye movement metrics between the two groups.

1.2 Outline

This chapter is organized as follows. Section 2 provides background on eye-tracking in general and as it has been applied specifically to program comprehension. Our experimental methodology is introduced in section 3 along with the metrics and data transformations used in our analysis. Section 5 presents a detailed analysis of our data, with section 5.2 breaking down results by program base. The discussion in section 6 connects the details of our results with the three research questions in this section. Lastly, section 7 concludes and considers future work.

2 Background

Although it has only recently been applied to program comprehension, eye-tracking has existing in one form or another for over a century. In this section, we start by reviewing the mechanisms of modern eye-trackers and the assumptions made when analyzing eye movement data. Next, we describe how eye-tracking has been incorporated into studies of program comprehension. Based on these studies, we outline expectations for the results of our experiment

2.1 Eye-tracking Methodology

Researchers have collected eye movement data for over one hundred years, usually to study natural language reading behavior [28]. Modern eye-trackers tend to use an infrared camera and LED to detect the pupil center and corneal reflection. With some calibration and a bit of trigonometry, an individual's "point of regard" can be determined fairly accurately [27]. Eye-tracking hardware can be broadly classified as **free-standing** or **head-mounted**. Free-standing eye-trackers function much like a webcam attached to a monitor. The participant is free to move about, though data accuracy is lost if their head moves too far outside of an

expected range [1]. Head-mounted eye-trackers solve this problem by attaching the camera to the participant's head, allowing more range of motion. This increases data accuracy, but may be distracting or otherwise interfere with the experiment.

Raw eye movement data, called gaze points, are processed into **fixations** and **saccades** by software outside the eye-tracker – typically provided by the hardware vendor. A fixation occurs when the eye maintains its gaze on a particular location for an extended period of time. Fixations last around a few hundred milliseconds, and are broken up by saccades: rapid jumps from one fixation location to another [2]. For a given task, fixations are mapped to specific **areas of interest** (AOIs): relevant regions of the task environment. AOIs are often individual words for reading tasks, or specific objects in a scene for image-based tasks. A series of fixations from one AOI to another is called a **scanpath**, and can be used to compare participants' task strategies [29].

A variety of eye-tracking metrics exist to quantify eye movement behavior. Fixation counts and duration for each AOI are common metrics for determining which words or objects participants considered most important. There is a crucial assumption being made here, called the **eye-mind hypothesis** – that point of regard and visual attention are strongly correlated. This hypothesis lies at the heart of most eye-tracking research, and is widely accepted by the eye-tracking community. It should still be kept in mind when interpreting results, however. Other eye movement metrics, such as the density of fixations in a given area or the amplitude of saccades, can be used to infer the quality of information cues in the task environment. All of these metrics, of course, depend on the details of the software used to transform gaze points into fixations and saccades. Like the eye-mind hypothesis, these details are potential threats to the validity of experimental results.

2.1.1 Benefits and Potential Problems

Eye movement data provides rich, fine-grained physiological data for visually-intensive tasks. With the eye-mind hypothesis, this allows researchers to gain a unique insight into a participant's cognitive processes with high temporal resolution. Other experimental protocols may provide similar insights, such as "think aloud" in which a participant narrates their activity during the task. The data from such protocols are necessarily mediated by consciousness, however, and therefore may not reflect the details of cognitive processes outside of conscious awareness [24]. To the extent that the task at hand depends on these types of processes, eye movements have an advantage. Additionally, the density of fixation and saccade data make it ideal for statistical methods, both within and between participants.

There are potential problems in the collection and interpretation of eye movement data. The accuracy and precision of fixations and saccades strongly depends on the eye-tracking hardware and experience of the data collector [25]. In some cases, a participant's eye color, contacts, and eye make-up have been found to decrease data accuracy. Even under ideal collection conditions, there is also some guesswork involved in the transformation of raw gaze data into fixations/saccades, and again in the mapping of fixations to areas of interest (AOIs). Using different methods for each data transformation step may lead to very different conclusions, especially when using a between-subjects experimental design with few participants

[1] A chin rest can help keep the participant's head still, but forces them to perform the experiment with a potentially unnatural posture.

[2] Besides alternating between fixations and saccades, humans and other animals have a "smooth pursuit" visual mode in which the eyes smoothly following a moving target [10].

(common in many eye-tracking studies). Finally, comparing eye movements between participants may require additional data massaging or quantization when looking behaviors are highly individualized. For example, the precise fixation order of code lines in our task varied considerably between participants, leading us to favor the comparison of high-level metric between groups of participants instead (see Section 5.1.3 for details).

2.2 Reading and Program Comprehension

Within the last decade, eye-tracking has become more prevalent in studies of *program comprehension*. In these studies, programmers perform a variety of tasks on source code, such as locating bugs [38], predicting output [18], and answering comprehension questions [8]. Participants' eye movements during program comprehension are used to infer the effects of visualizations [1], expertise [13], pair programming [33], and other environmental factors.

Bednarik et. al performed extensive eye-tracking experiments with expert and novice programmers, focusing on the utility of program visualizations alongside source code [1]. Besides task performance, they found differences between novices and experts in both low-level eye movement metrics (mean fixation duration), and high-level looking behaviors (attention switches between code and visualization). Experts, for example, attended to the source code more than novices, and appeared to integrate more information between the different representations of a program. An earlier paper by Bednarik also found that experts were more affected by being force to use a restricted focus viewer (RFV) when viewing a program's source code and alternative graphical representation [3]. The RFV blurred the contents of the screen except for a small region controlled by the participant. In this paper, Bednarik et. al found that, while task performance did not decrease, more experienced programmers switched visual attention between representations less often when the RFV was active. Novices did not modify their behavior, however, suggesting that experts were making heavier use of peripheral vision when RFV was not active.

Uwano et. al had participants review C source code to locate defects, and analyzed the resulting patterns in their eye movements [38]. They observed a particular pattern, called *scan*, which represents a preliminary reading of the entire program (12-23 lines long) from top to bottom. In their experiment, approximately 73% of code lines tended to be fixated within the first 30% of a trial. Participants who spent more time in this scan pattern were also more likely to detect the defects in the program. We observed a similar scan pattern during some trials, but did not find a correlation with task performance.

Busjahn et. al compared the behavior of programmers reading both Java source code and natural language texts [8]. Significant differences were found in low-level eye movement metrics between text types, such as mean fixation duration and regression rates [3]. Substantial variation was also found between participants and between texts of the same type (source code or natural language). Additionally, the proportional fixation times for different categories of "words" in source code was analyzed, normalized by number of characters [4]. Out of keywords, identifiers, numbers, and operators, Busjahn et. al found that keywords received the least amount of fixation time per character. An earlier study by Crosby found similar results for keywords, though their other source code categories differed [13]. We observed the same pattern for keywords, and found that

[3] A regression occurs when a previously fixated word is fixated again.

[4] As noted by Busjahn et. al, normalizing by syllables or some other unit that can be processed within a single fixation may be more appropriate.

lines with mathematical operators or comparisons received the most fixation time (Section 5.1.2).

2.2.1 Expectations for Our Data

Based on the body of eye-tracking program comprehension study literature, we can form expectations for our own experiment. These expectations may be violated, however, given the uniqueness of our task. As mentioned previously, many eye-tracking experiments in this area ask participants to debug or answer comprehension questions about programs. In contrast, our participants were tasked with predicting the precise printed output of a program. Within the cognitive modeling subsection of program comprehension, programmers have demonstrated significant differences in their recalled representations of programs based on the task at hand [14]. Specifically, a programmer's "mental model" will differ if they're asked to recall relevant portions of a program for the purposes of documenting or summarizing versus modifying or reusing it. An analogous difference may be expected when programmers are asked to answer comprehension questions about, or literally evaluate, a program's source code.

Assuming *some* similarity between our task and others, however, we should expect mean fixation time to be related to experience [1], and to fall within the 300-400 ms range [8]. Regression rates are also predicted to be in the 30% to 40% range, well above the typical 10% to 15% range for natural language text [28]. When analyzing reading behavior, we should also expect the majority of lines to be fixated *in line order* within the first 30% of the trial [38]. Finally, participants are expected to fixate on more "complex" statements about twice as much as "simple" statements, and to spend the least amount of time on keywords [13].

Next, section 3 introduces our experimental methodology. Section 5 then delves into the details of our results. We analyze participants' eye movements in terms of reading behavior, by program base/version, and by participant demographics. Our data conformed to some of the expectations in this section (e.g., keyword fixation time), but violated others (e.g., fixation duration and experience). The discussion in section 6 considers reasons for these violations, and relates the detailed results to our research questions.

3 Methodology

We now introduce our experiment design and eye-tracking hardware. Section 3.3 describes the process of transforming raw eye movement data into fixations, and mapping those fixations to areas of interest. Lastly, section 3.4 provides definitions for each of the eye movement metrics used in the analysis.

3.1 Experiment and Participant Demographics

We recruited 29 via e-mail and from an introductory programming class at Indiana University. Participants were paid $10 each, and performed the experiment in front of an eye-tracker. All participants were screened for a minimum competency in Python by passing a basic language test. The mean participant age was 27.3 years, with an average of 2.9 years of self-reported Python experience and 9.4 years of programming experience overall. Most of the participants had a college degree (86.2%), and were current or former Computer Science majors (65.5%). Figure 1 has a more detailed breakdown of the participant demographics.

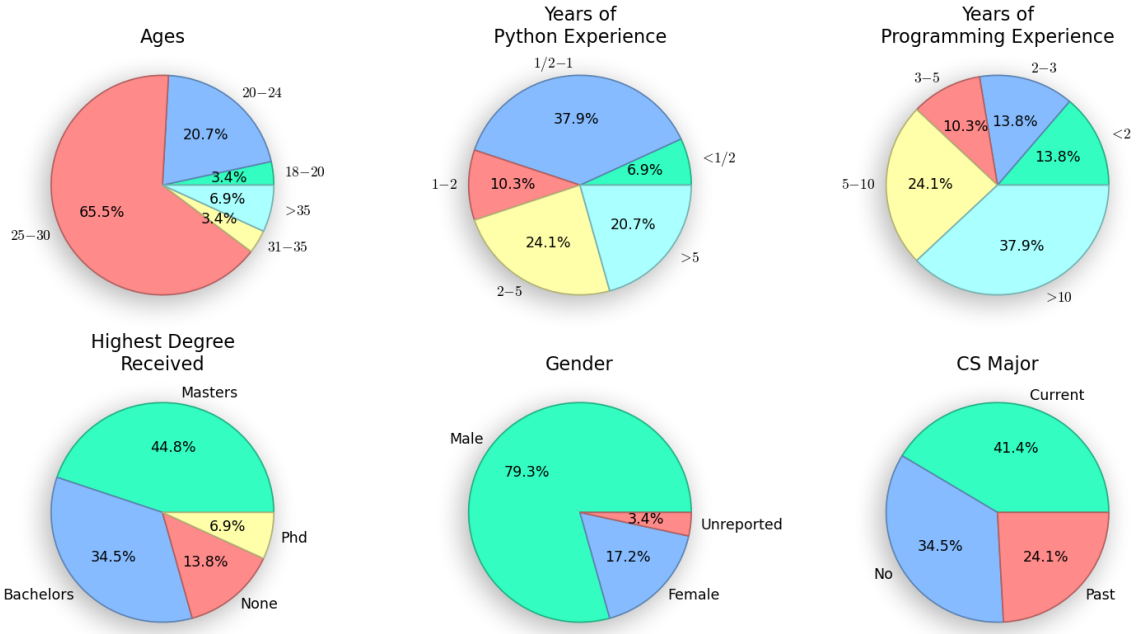


Figure 1: *Demographics of all 29 participants.*

The experiment consisted of a pre-test survey with questions about demographics and experience, ten trials (one program each), and a post-test survey assessing confidence and requesting feedback. The pre-test survey gathered information about the participant's age, gender, education, Python experience, and overall programming experience. Participants were then asked to predict the printed output of ten short Python programs, one version randomly chosen from each of ten program bases (Figure 2). The presentation order and names of the programs were randomized, and all answers were final. No feedback about correctness was provided and, although every program produced error-free output, participants were not informed

of this fact beforehand. The post-test survey gauged a participant's confidence in their answers and the perceived difficulty of the task overall.

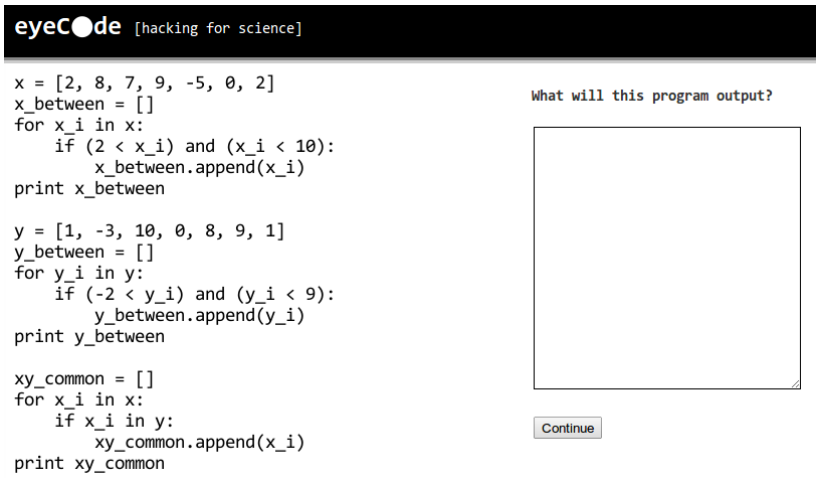


Figure 2: *Sample trial from the experiment (`between inline`). Participants were asked to predict the exact output of ten Python programs.*

We collected a total of 288 trials from 29 participants starting November 20, 2012 and ending January 19, 2013. Trial responses were manually screened, and a total of 2 trials were excluded based on the response text. Participants were not constrained to complete individual trials or the experiment in any specific amount of time. There were a total of twenty-five Python programs in our experiment belonging to ten different program bases. The programs ranged in size from 3 to 24 lines of code, and did not make use of any standard or third-party libraries.

3.2 Eye-Tracking Hardware

The Tobii TX300 is a free-standing eye-tracker that collects gaze data at 300Hz, and averages an error of around 0.04 degrees of visual angle for both eyes [5]. The screen is 23 inches in size and measures 557 mm across with a 1920x1080 maximum resolution. Before starting the experiment, participants went through a brief calibration using the vendor-provided Tobii Studio software package. Aside from being asked to sit about 65 cm away from the eye-tracker and not to move their heads as much as possible, participants interacted with the TX300 as if it were a normal computer.

We used Tobii Studio 2.2, a software package provided by Tobii with the eye-tracker, to do calibration and to record/process raw gaze data into fixations and saccades. The TX300 records raw moment-to-moment gaze points, which Tobii Studio collects and processes using a *fixation filter*. There is no precise means of translating gaze points into fixations, so we relied on Tobii Studio's default I-VT fixation filter to perform this task for us. The Tobii Studio user manual [36] describes this filter as follows:

> *The general idea behind an I-VT filter is to classify eye movements based on the velocity of the directional shifts of the eye. The velocity is most commonly given in visual degrees per second (°/s). If the velocity of*

[5] Note, however, that anything within 1 degree may be fixated by the fovea without moving the eyes.

Figure 3: *Tobii TX300 Eye-tracker. Screen size is 23 inches (557 mm wide) with 1920x1080 resolution. When a participant is seated 65 cm away from the screen, it will subtend about a 31 degree visual angle.*

the eye movement is below a certain threshold the samples are classified as part of a fixation.

More technical information about the I-VT fixation filter can be found in a Tobii whitepaper [26]. Per the Tobii Studio user manual, we also filtered out fixations which the eye-tracker marked as potentially invalid (i.e., with a left or right eye validity code higher than 1).

Over the course of 290 trials (10 trials per participant), we collected around 50,000 fixations (average of 172 per trial). Each fixation consists of a two-dimensional screen coordinate, a start time, and a duration in milliseconds. A quick plot of one trial's fixations on a static image of the participant's screen immediately reveals one of the many challenges involved in analyzing gaze data (see Figure 4). Like the gaze point to fixation translation step that Tobii Studio's fixation filter performs, there is some guesswork involved in mapping fixations to *areas of interest* (AOIs). In the next section, we describe our method for assigning fixations to words, lines, and interface regions on the screen.

3.3 Areas of Interest

An area of interest (AOI) is a region of the screen where we would like to know when the participant is fixating. We define three kinds of code AOIs (block, line, syntax) and two kinds of interface AOIs (output box, continue button). A **block** AOI is one or more lines of code separated from other blocks by at least one blank line. Figure 5 shows the block AOIs for the `between_functions` program (highlighted on the left side of the screen) as well as the **output box** and **continue button** AOIs (highlighted on the right side of the screen). Figure 6 shows the **line** and **syntax** AOIs for the same program. Line AOIs include indentation because it is semantically relevant to Python. Syntax AOIs are automatically computed with the popular Pygments library [5] and assigned one of the following categories: keyword, identifier, operator (+, −), literal, or comparison (<, >).

Because fixations are essentially timestamped screen coordinates, the easiest way to map a fixation to an AOI is to simply check if the fixation point lies within one of the AOI rectangles (Figure 7). This method, which we will refer to as **point mapping**, is fast and easy to compute. When AOIs are large (relative to the

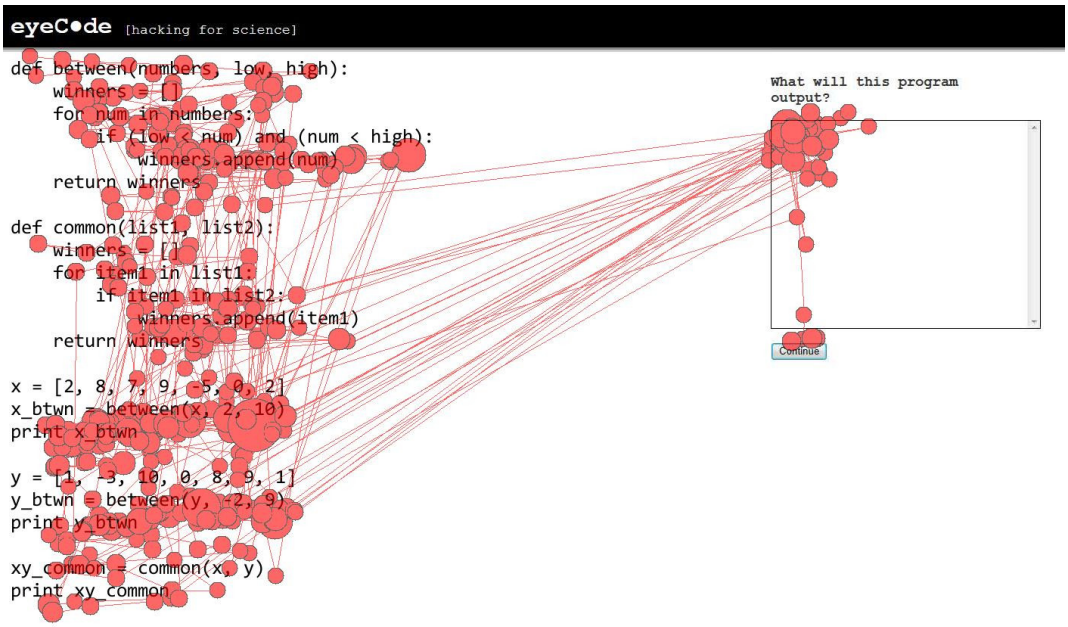


Figure 4: *Fixation circle plot for a single trial (`between_functions` program). Circle radii are proportional to fixation duration.*

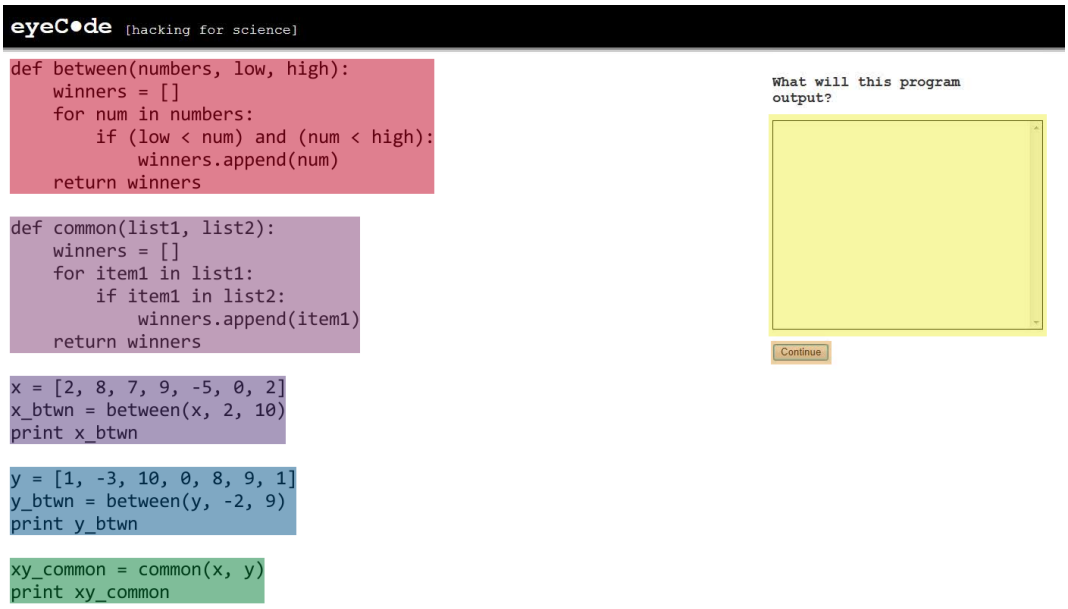
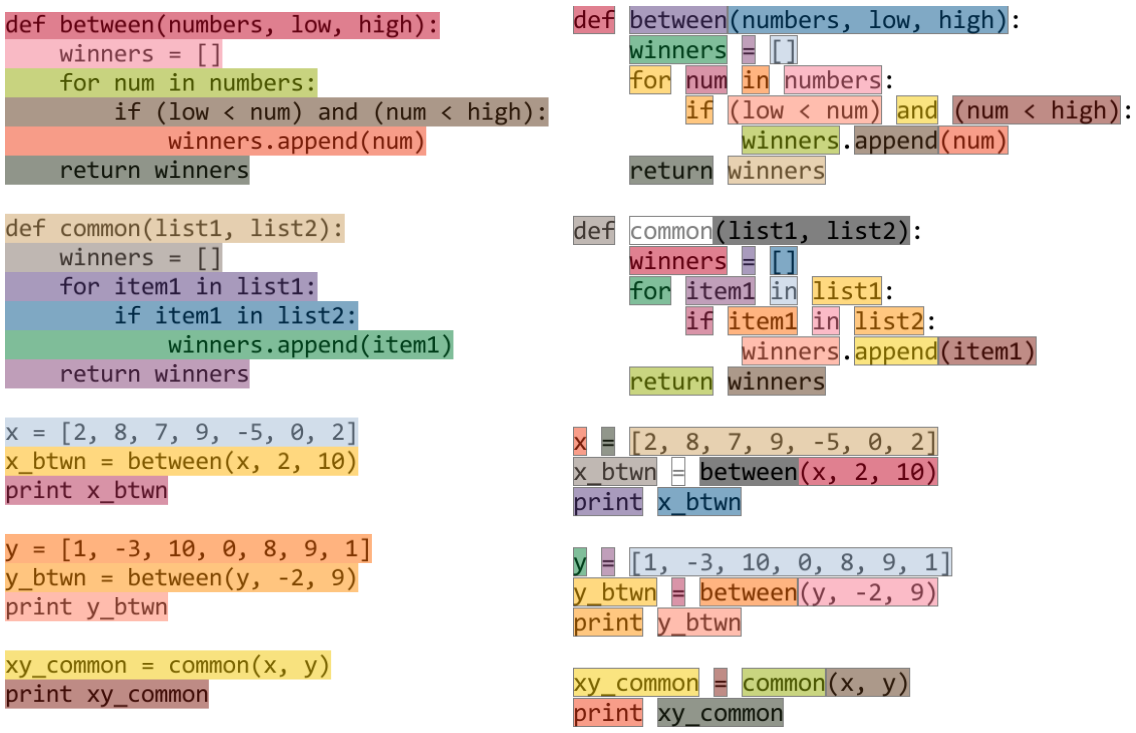


Figure 5: *Block areas of interest for the (`between_functions` program). The output box and continue button are also areas of interest.*



```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

def common(list1, list2):
    winners = []
    for item1 in list1:
        if item1 in list2:
            winners.append(item1)
    return winners

x = [2, 8, 7, 9, -5, 0, 2]
x_btwn = between(x, 2, 10)
print x_btwn

y = [1, -3, 10, 0, 8, 9, 1]
y_btwn = between(y, -2, 9)
print y_btwn

xy_common = common(x, y)
print xy_common
```

```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

def common(list1, list2):
    winners = []
    for item1 in list1:
        if item1 in list2:
            winners.append(item1)
    return winners

x = [2, 8, 7, 9, -5, 0, 2]
x_btwn = between(x, 2, 10)
print x_btwn

y = [1, -3, 10, 0, 8, 9, 1]
y_btwn = between(y, -2, 9)
print y_btwn

xy_common = common(x, y)
print xy_common
```

Figure 6: ***Left:*** *line areas of interest in the* `between_functions` *program.* ***Right:*** *syntax areas of interest in the* `between_functions` *program.*

expected error of the eye-tracker) and separated by a sufficient amount of empty space, point mapping can be used to accurately map fixations to the correct AOIs. If AOIs are small and close together, however, point mapping may fail to correctly map fixations due to hardware error or participant movement. Extending the boundaries of an AOI can help, but only if one AOI rectangle does not overlap another. Because our code AOIs are relatively small and close together, we consider a second mapping method that we call **circle mapping**.

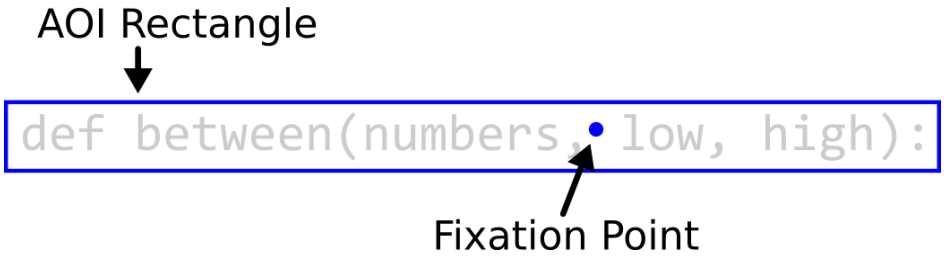


Figure 7: *Point-based AOI mapping. Fixations are mapped to an AOI if the fixation point lies within the AOI rectangle.*

When circle mapping fixations to AOIs, a circle is centered around the fixation point. For our analysis, the radius of the fixation circle (20 pixels) was chosen based on the size of our text and expected error of the eye-tracker. The area of the circle's intersection with every AOI is computed (Figure 8), and the AOI with the largest area of overlap is mapped to the fixation. This mapping method is more computationally intensive, but has an advantage over point mapping when AOIs are close and their boundaries cannot be extended without overlap. When a fixation occurs near two AOIs, the closest one will be mapped to the fixation. Circle mapping could also be used to assign probabilities to multiple AOIs (based on area of overlap), and combined with a Markov model to find the most likely series of fixations over time (see Future Work).

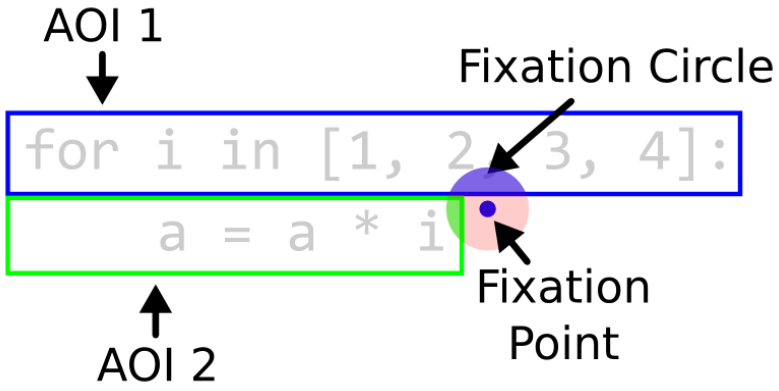


Figure 8: *Circle-based AOI mapping. Fixations are surrounded by circles and mapped to the AOI with the biggest area of overlap (AOI 1 in this case).*

3.3.1 Offset Correction

During the data cleaning step of our analysis, fixation coordinates were manually adjusted in the vertical direction for each participant. A *single vertical offset* was used to correct fixations for each participant's trials, and was chosen to increase the overlap of fixations with non-empty portions of the screen. Figure 9 demonstrates the process: an uncorrected fixation plot is shown on the left. The fixations near the continue button (highlighted in green) suggest that the eye-tracker may be reporting gaze points slightly low for this participant. On the right, the corrected fixation plot lines up better visually with the non-empty portions of the screen. All offsets are documented, and the original raw data has been preserved.

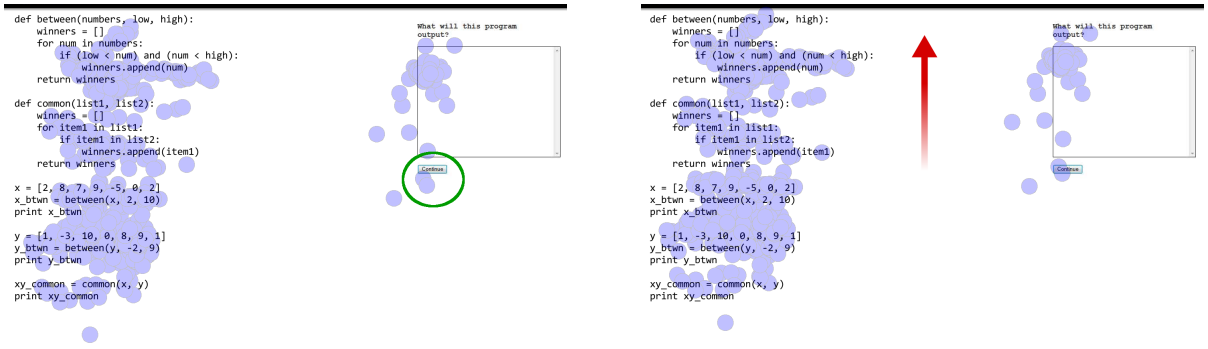


Figure 9: *Example of offset correction. Raw fixations (left) were shifted up to better align with known areas of interest (right).*

While an automated offset correction process would have been preferred, conventional methods require knowing where a participant **must** be looking at some point in time. We did not require that participants fixate in a specific spot at the start or end of a trial, so this information was not available. Although it would be reasonable to expect participants to fixate on the continue button before ending a trial, we found several instances where this was clearly not the case. Indeed, some participants hovered their mouse cursor over the continue button, shifted visual attention to the code (perhaps for a final check), and clicked the button without re-fixating it first!

3.3.2 Scanpath Comparisons

With fixations mapped to areas of interest, we can now codify a participant's behavior over time. A **scanpath** is a string representing the order in which AOIs were fixated during a trial. This string does not typically contain any duration information, and adjacent fixations on the same AOIs are usually collapsed into a single symbol. For example, the sequence of fixated AOIs in Figure 10 is `AABCBCCA`. Removing duplicate adjacent fixations, we are left with the scanpath `ABCBCA`.

Scanpaths can be used to generate **transition matrices**, which quantify how often one AOI is fixated after another. Using the scanpath from our example above (`ABCBCA`), the transition matrix in Figure 11 shows that `B` always follows `A`, that `C` always follows `B`, and that both `A` and `B` follow `C` half the time. More sophisticated methods, such as the scanpath successor representation [19] incorporate discounted temporal information (e.g., `C` *eventually* follows `A`). For our analysis, however, we only make use of simple transition matrices.

Metrics like the Levenshtein Distance (commonly know as string edit distance) can be used to compare

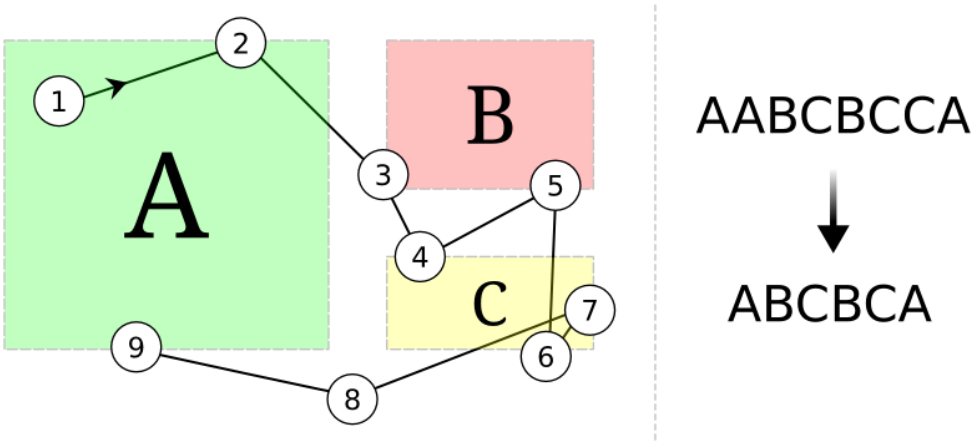


Figure 10: *Creation of an AOI scanpath. Fixations are mapped to an AOI sequence string, often with repeated items removed.*

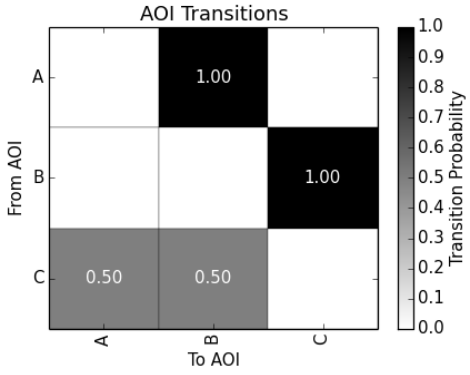


Figure 11: *Transition matrix for the scanpath* `ABCBCA`*.* `B` *always follows* `A`*,* `A` *follows* `C` *half the time, etc.*

scanpaths. This distance metric between two strings increases by one for each insertion, deletion, or character change necessary to transform one string into another [30]. The distance between similar strings, such as `123` and `1234` will be small (in this case, just 1). The maximum edit distance between two strings is the length of the largest string, and thus we can use this value as a normalization factor (distance / length of largest string). The normalized edit distance between scanpaths can be used to cluster trials by **strategy** – common orderings of AOI fixations. This makes sense when there are strong expectations about AOI fixation order, and when scanpaths are close in length. More general algorithms, such as ScanMatch [12], make use of techniques developed for DNA sequence matching. These algorithms allow for more fine-grained control of sequence substitutions (e.g., by penalizing specific character replacements). They are also much better at dealing with scanpaths that have missing segments.

3.4 Eye-Tracking Metrics

Once gaze data has been processed into fixations and saccades, there are a plethora of metrics available to summarize the data [27]. Table 1 lists the handful of metrics we used to summarize participant fixations within and between trials. The **fixation count** is simply the number of fixations, while **mean fixation duration** is the average amount of time a participant spent fixation some area of interest (AOI). **Fixation rate** is the number of fixations per second over the course of a trial. **Spatial density** divides an AOI into a grid, and is higher when more cells in the grid receive at least one fixation. The **normalized saccade length** and **average saccade length** measure the total and average Euclidean distances between fixations. Finally, the **Uwano review percent** is the percentage of lines fixated in the first 30% of a trial.

3.4.1 Line Metrics

What properties of a line of code influence a participant's reading behavior? We might expect the position and length of a line to influence when a participant fixates the line and for how long. We define several **textual** and **content-based** metrics to quantify the properties of each line of code. Table 2 describes textual metrics, which could be applied to any text – source code or natural language. These metrics quantify the size and position of a line, as well as the distribution of whitespace within it. Table 3 lists metrics specific to source code, such as the number of keywords and operators.

Name	Description
Fixation Count	Number of fixations in a given time period.
Mean Fixation Duration	Total fixation duration in a given time period divided by the fixation count.
Fixation Rate	Number of fixations divided by the number of seconds in a given time period.
Spatial Density	Number of grid cells containing at least one fixation in a regular $N \times N$ grid covering a given AOI [11].
Normalized Scanpath Length	Total Euclidean distance between fixations in a scanpath divided by time between first and last fixation.
Average Saccade Length	Average euclidean distance between start and end points of each saccade.
Uwano Review Percent	Percentage of code lines reviewed within the first 30% of a trial [38]

Table 1: *Eye-tracking metrics used in our analysis.*

Textual Metric	Definition
Line length	Number of characters (excluding indentation)
Line number proportion	Line number divided by total number of lines
Whitespace proportion	Number of spaces (excluding indentation)
Indentation level	Number of 4-space blocks on the left

Table 2: *Textual line metrics*

Content Metric	Definition
Keyword count	Number of `class`, `def`, `for`, `if`, `print`, `return`
Operator count	Number of `*`, `+`, `-`, `.`, $<$, `=`, $>$, `and`, `in`
Identifier count	Number of class/function/variable names
Line category	Content of line, one of: • List creation (literal $[1, 2, 3]$) • Comparison ($x < y$) • Math operation ($+$, $*$, $-$) • For loop (`for x in y`) • Function call (`f(x)`) • Function definition (`def f(x)`) • If statement (`if x`) • Print statement (`print x`) • Return statement (`return x`) • Class definition (`class Foo`) • Assignment (`x = y`)

Table 3: *Content-based line metrics*

The **line length** is simply the number of characters in a line, excluding the whitespace at the start of a line. Rather than use line number, we calculate **line number proportion**. This metric tends to correlate better with the time of first fixation (Section 5.1), suggesting that participants perform an initial scan of a program faster for larger programs. The **whitespace proportion** of a line is simply the proportion of spaces to characters, excluding initial indentation. Lastly, the **indentation level** measures the amount of whitespace before a line begins, divided into 4 space blocks (a standard in Python).

For content-based metrics, we define the **keyword count**, **identifier count**, and **operator count**. These are simply the number of keywords, variable/function names, and mathematical/boolean operators in a given line. The **category** of a line was based on a number of factors. For example, lines containing a `for` or `return` keyword were classified as "For Loop" and "Return Statement." Some lines, such as `x = [1, 2, 3]`, could have multiple categories (assignment, list creation). We picked the category for each line that was highest in the list given in the final cell of Table 3, so the example line would be classified as a list creation rather than an assignment.

4 The eyeCode Library

To facilitate the analysis of eye movement data in the context of program comprehension, we created a Python library called **eyeCode**. Tobii Studio provides basic plots and statistics, but customization is limited to a few menu options. A more sophisticated tool, OGAMA (Open Gaze and Mouse Analyzer), is a freeware package with many more bells and whistles [39]. In addition to standard eye movement plots, OGAMA can be used to create areas of interest and compute the Levenshtein distance between scanpaths. Like Tobii Studio, however, OGAMA is intended to be fairly task neutral, and therefore did not have specific tools for program comprehension experiments.

The eyeCode library is built on top of the `pandas` statistical computing library [23], and contains specialized functions for processing, plotting, and computing metrics over fixations and saccades on a static code display. The source code is freely available at `http://github.com/synesthesiam/eyecode` under a liberal open source license. Example analyses and data from multiple experiments (including this one) are embedded into the library along with participant info and a web-based fixation viewer.

4.1 Areas of Interest

Identifying areas of interest (AOIs) can be done **automatically** for programming languages that are supported by the popular Pygments library [5]. This relieves the researcher of the need to manually create AOIs for each code "word", line, and block. Pygments contains lexers for a variety of languages and, when combined with a monospace font size and line spacing, can be used to generate rectangles for every token in a program. Figure 12 shows the token AOIs identified from a short Java program. Tokens are then merged into lines, and lines are merged into whitespace-separated blocks.(Figure 13). For experiments using variable-width fonts, eyeCode has methods for scanning raw images to locate words and lines.

```
package hello;

class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Figure 12: *Areas of interest for a Java program automatically identified using the Pygments Python library.*

After identifying areas of interest, eyeCode can map fixations to AOIs via a process called **hit testing**. Two methods are currently supported: point mapping and circle mapping (see Section 3 for details). AOIs in eyeCode can also be grouped into layers, allowing overlapping AOIs to be hit-tested simultaneously (only AOIs within a layer must be disjoint). Hit-tested fixations can be easily converted to scanpaths, summarized with metrics, or visualized with one of many plots.

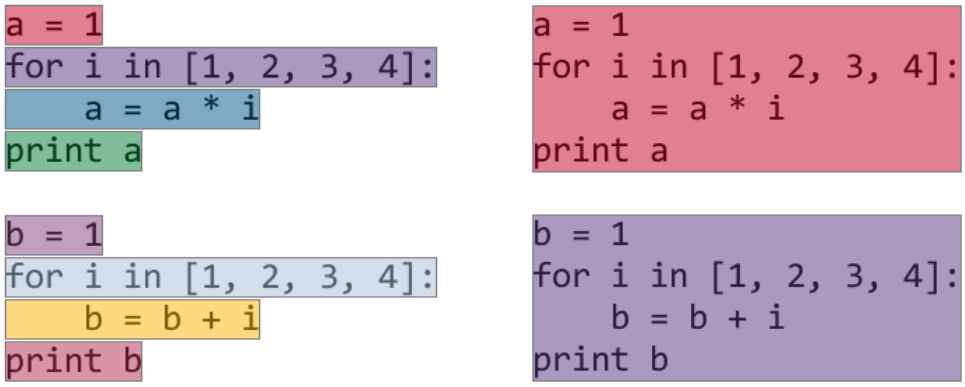


Figure 13: ***Left****: Line-based AOIs for the* `initvar - onebad` *program.* ***Right****: Block-based AOIs for the same program.*

4.2 Metrics and Plots

Common eye movement metrics, such as fixation duration, can be computed per area of interest and across trials (Figure 14). Higher-level metrics, like transition matrices and spatial density [11], are available along with many custom visualizations. Because lines are especially relevant for code, eyeCode contains specialized plotting functions for both static and dynamic views of line fixations across and within trials. Figure 15 shows an aggregate view of fixation duration by line for all participants in a single program (top), and a timeline of fixations on each line for a single trial (bottom). Fixations in the timeline above the dotted line occurred in the output box – the text area where participants enter their predictions of program output.

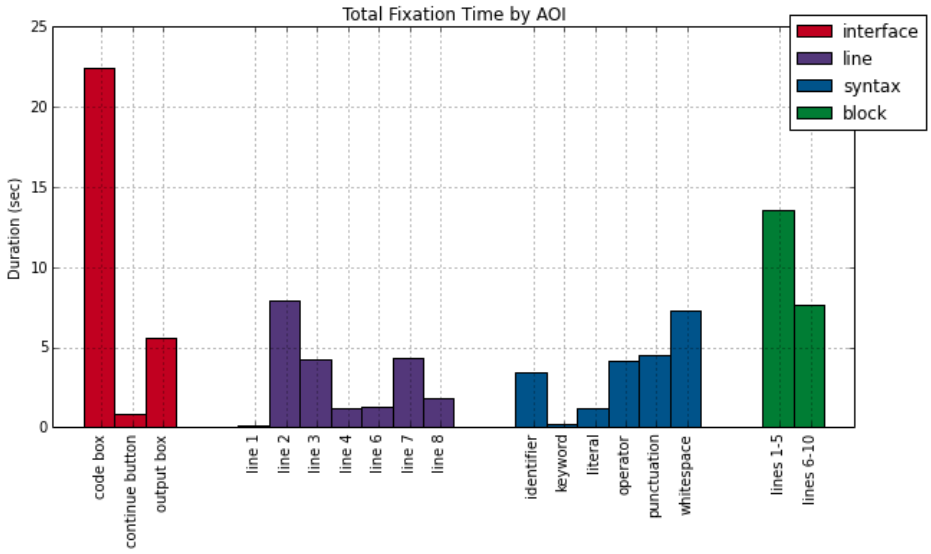


Figure 14: *Total fixation duration by AOI kind and name for* `initvar - onebad` *(all participants).*

The "Super Code" plot combines fixation duration information about code elements, lines, and blocks into a single plot (Figure 16). The color of each code element represents the relative amount of time spent on

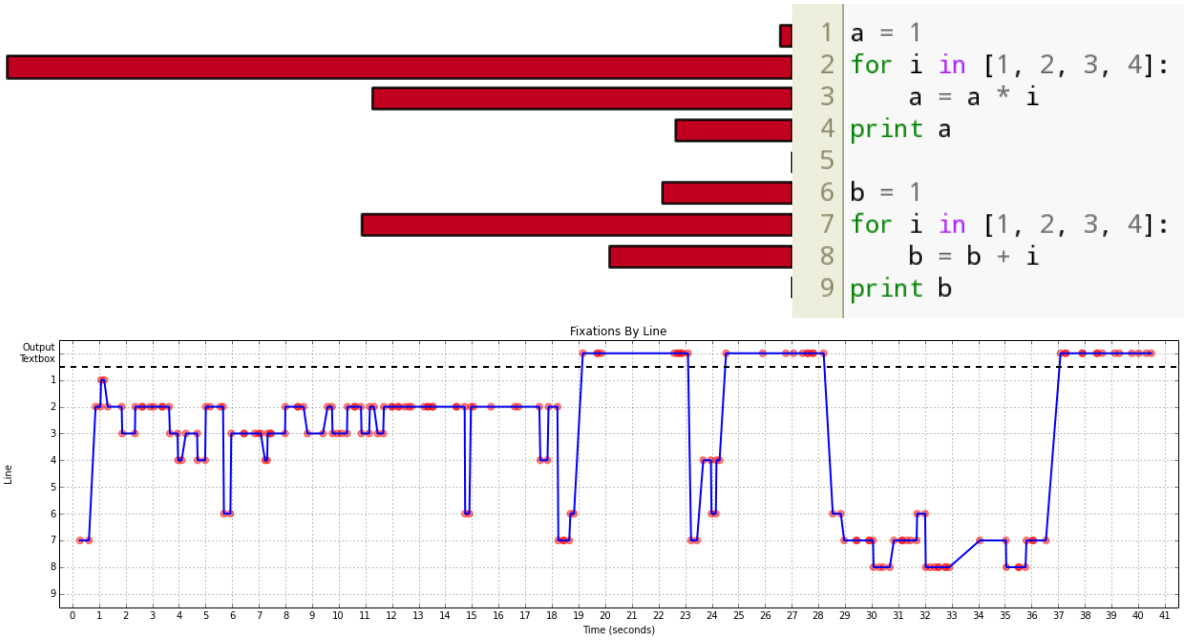


Figure 15: ***Top****: Total fixation duration for each line (all participants).* ***Bottom****: Fixation timeline for experiment 1 trial 1.*

that element. Each line has a bar next to it, with the length proportional to the total fixation duration on that line. These bars are also colored, and each block of code shares the same color. This shows the relative amount of time spent on each block as a whole – the darkest red set of bars had the most fixation duration.

4.3 Rolling Metrics

The eyeCode library supports computing some low and high-level metrics over a rolling time window in a trial. Comparing the visualization of these rolling metrics with the line fixation timeline allows for the gathering of additional evidence to support hypothesis about participants' real-time cognitive processes. In Figure 17, for example, both the average length of a saccade and average duration of a fixation are computed every half second across a one second window. Large changes in either metric can be mapped back to the line fixation timeline in Figure 15, and potential causes can be enumerated based on where and when the participant is looking. Some metric spikes, such as the increase just after 25 seconds, may be expected – here, the participant is clearly transitioning back and forth between the output box and the source code. Others, such as the increase in average fixation duration just after 15 seconds, suggest increased focus. Given the subsequent transition to the output box (and that the first response characters are typed afterwards), it's reasonable to guess that the participant is mentally calculating the product $1 * 2 * 3 * 4 = 24$. Indeed, inspecting the participant's response reveals an "a = 24" following by a correction to just "24".

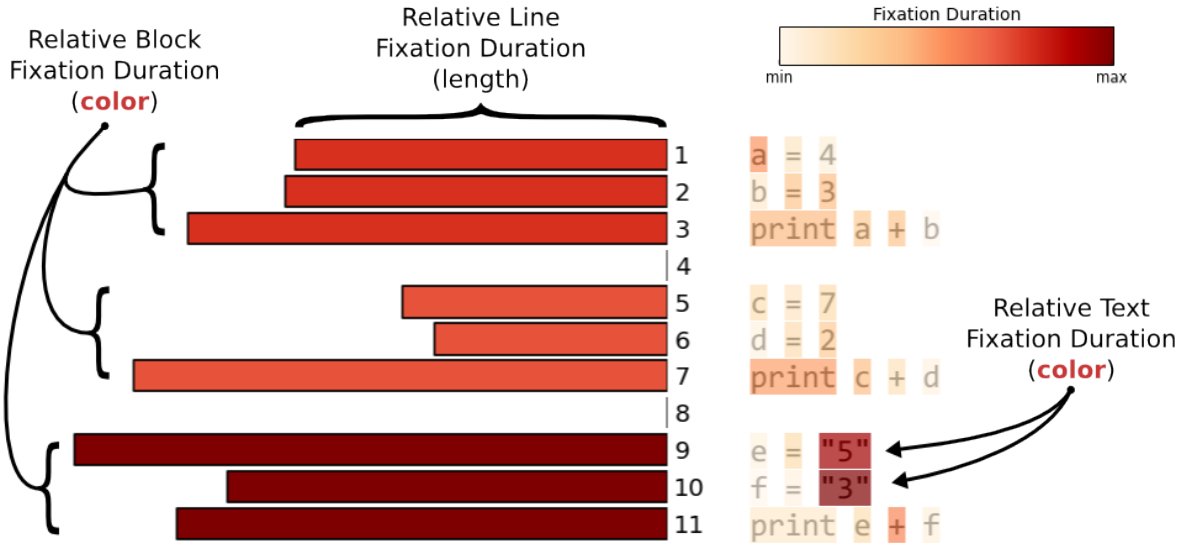


Figure 16: *Relative fixation duration plot for all* `overload plusmixed` *trials. Text color indicates fixation duration relative to other code elements. Bar color indicates relative fixation duration per whitespace-separated block. Bar length is proportional to within-block fixation duration.*

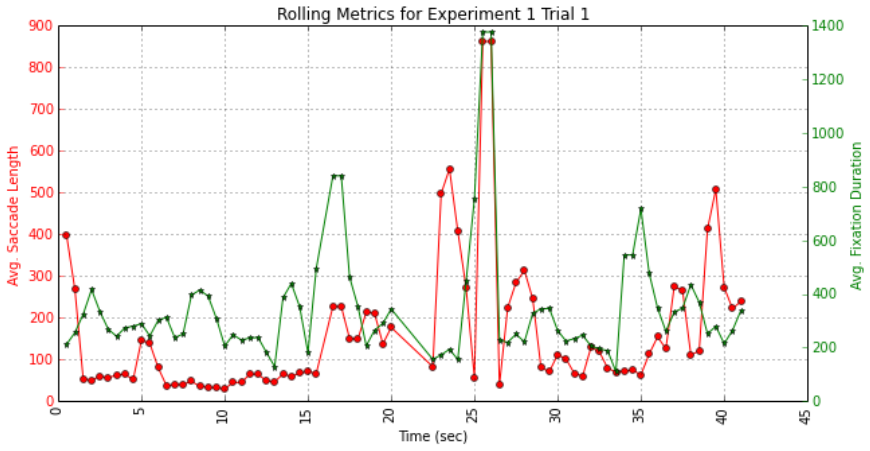


Figure 17: *Rolling metrics for experiment 1 trial 1 (*`initvar - onebad`*). Average saccade length (red, left) and average fixation duration (green, right) computed over a one second window every half second.*

5 Results

We collected approximately 50,000 fixations from 288 trials across 29 participants. Below, we analyze and summarize our data first across all trials (Section 5.1), and then by individual program kind (Section 5.2). Plots and analyses were done using the eyeCode library. Statistics were computed with `scipy` [21], and linear models were fit using the `statsmodels` package [32].

Non-parametric statistics were used to compare groups (Mann-Whitney U test) and determine correlations (Spearman's r) [34]. We calculate effect sizes for U tests using the rank-biserial correlation r, a metric whose range is $[-1, 1]$ with 0 meaning no correlation [40]. As a rule of thumb, we take absolute values of r greater than or equal to 0.2 to indicate a meaningful relationship (and $|r| > 0.4$ as a strong relationship). For correlations, we infer weak, moderate, strong, and very strong relationships when $|r|$ is greater than or equal to 0.2, 0.3, 0.4, and 0.7 respectively.

5.1 Reading Behavior

Across all participants and trials, we observed a mean fixation duration of 273 ms. Fixations typically last 200-300 ms [28], but higher values have been observed in several program comprehension studies. A fixation duration range of 309-408 ms was found in a program comprehension study by Busjahn et. al [8]. Participants in the same study read natural language texts, and a typical fixation duration range of 232-285 ms was recorded. In another study by Bednarik et. al, participants under most conditions had mean fixation durations in the 300-400 ms range when viewing code [1].

When fixating the output box – the text box into which participants typed their responses – fixations were significantly longer, with an average of 331 ms versus 267 ms when fixating source code ($U = 89197473.5$, $p < .001$, $r = 0.03$). This suggests that participants were attending to their own predicted output slightly more, but the effect size is too small to be sure. Per trial, we observed an average of 172 fixations and approximately 2.73 fixations per second (Figure 18). At the trial level, and for each line of code, total fixation count and duration were almost perfectly correlated ($r = 0.97$, $p < .001$). At the interface level, participants produced an average of six transitions between the source code and output box, indicating an on-demand construction of responses. Our analyses of individual programs' code line and output box transition matrices support this hypothesis by correlating output transitions and responses.

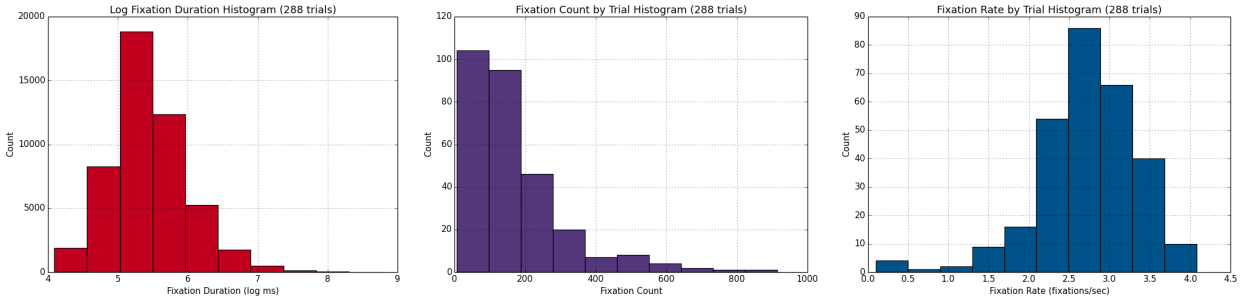


Figure 18: *Fixation metric distributions for all trials.*

Participants tended to read programs in line order (see Figure 19 for an example). The time of their

first fixation (proportional to the trial duration) on each line was strongly correlated with the line number (proportional to the total number of lines in the program). In fact, line number proportion alone accounts for approximately half of the variance in a simple linear model predicting first fixation proportion ($r^2 = 0.57$). Textual line metrics, such as line length, indentation, and whitespace proportion did not significantly improve predictions. Additionally, content-based line metrics (e.g., number of identifiers, operators) did not improve the linear model's performance.

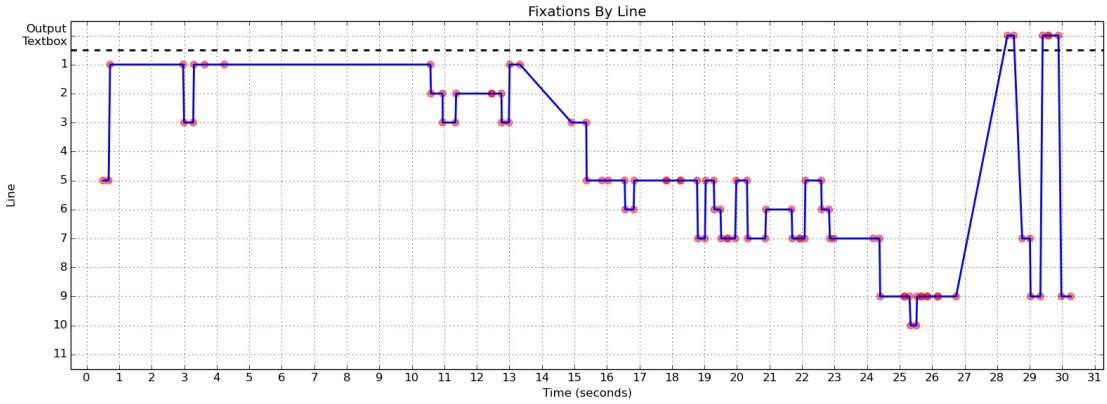


Figure 19: *Timeline for trial 268 (`overload plusmixed`). Participants read programs (mostly) in line order before responding.*

Longer lines tended to receive more time: total fixation duration per line and line length were strongly correlated ($r = 0.41$). Adding the whitespace proportion and length of the line to a simple linear model predicting fixation duration produces an r^2 of 0.54. Surprisingly, additional textual and content-based line metrics to not improve model performance significantly. This may be due to "important" program lines simply being longer, such as lists containing values necessary for calculations. The relationship between category and fixation duration may also be non-linear, resulting in a plateau in linear model performance.

While the category of a line was not a significant predictor of first fixation or total fixation duration, there are potential confounds due to patterns in the our source code. Across all 25 programs, whitespace proportion decreased with line number (i.e., textual density increased). Additionally, operator counts were positively correlated with whitespace proportion and identifiers, but negatively correlated with keyword counts. These patterns may have induced co-variances between textual and content-based metrics, which resulted in a lack of model performance improvement. A meta-study of multiple code repositories would be required to determine whether these patterns hold in general across Python code, or if they are simply an artifact of the programs in our experiment.

Based on other eye-tracking studies, we expected to find a correlation between programmers' experience level and eye movement metrics, especially mean fixation duration. Each participant's distribution of mean fixation durations per trial is shown in Figure 20. While participants are somewhat distinguishable by these distributions, they are heavily skewed. In addition, sorting by years of programming experience on the x-axis does not show evidence of a strong trend. A linear model predicting mean fixation duration from programming experience *does* produce a significant p-value (< 0.05) and a negative coefficient, but the effect size is extremely small ($r^2 = 0.021$).

A similar pattern is seen when looking at the number of fixations per second (fixation rate). Figure 21

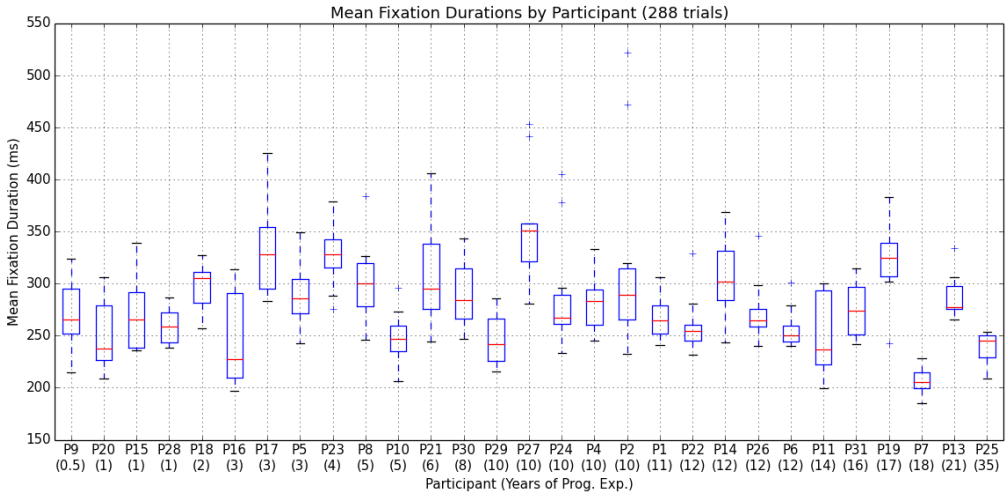


Figure 20: *Mean fixation duration for each trial, grouped by participant. Participants are ordered from least to most programming experience.*

shows the distributions of mean fixation rates by trial for all participants. The x-axis is again sorted by years of programming experience, but this time there is not even a statistically-relevant trend. The discussion in section 6 provides potential reasons for the mismatch between experience and fixation duration/rate – our unique task and fairly simple programs.

Uwano Review Percent and Spatial Density. Uwano et. al found that programmers tended to fixate 72.8% of code lines during the first 30% of a trial [38]. We refer to this value as the **Uwano Review Percent**. Interestingly, we had a very similar observation: across all trials, an average of 72.5% of code lines were fixated in the first 30% of a trial. However, this number appears to be largely driven by the number of lines in the trial's program. Percentages ranged from 44.4% to 91.1% depending on the program base and version. Additionally, a strong correlation was found between the Uwano Review Percent and the number of lines of code for individual trials ($r = -0.40$, $p < 0.05$). Thus, we cannot infer something general about a programmer's behavior, independent of the program, from the Uwano Review Percent.

We computed the **spatial density** of fixations over an envelope surrounding all lines of code for each trial. This envelope was divided into a grid whose cells were 30 pixels by 30 pixels, and the density was simply the percentage of these cells that had at least one fixation [11]. Across all programs, the mean spatial density was surprisingly consistent with a mean of 0.40 and a standard deviation of 0.07. This value is strongly correlated with the percentage of the code area that has text in it, however, suggesting the obvious – participants are not fixating empty space ($r = 0.57$, $p < 0.01$). Different grid sizes may provide more useful information, but we will leave this exercise for future work.

5.1.1 Saccade Angles

Saccades are the quick jumps between fixations, and can reveal interesting details about a task. The angle and distance between the source and destination fixation of each saccade can be plotted and compared

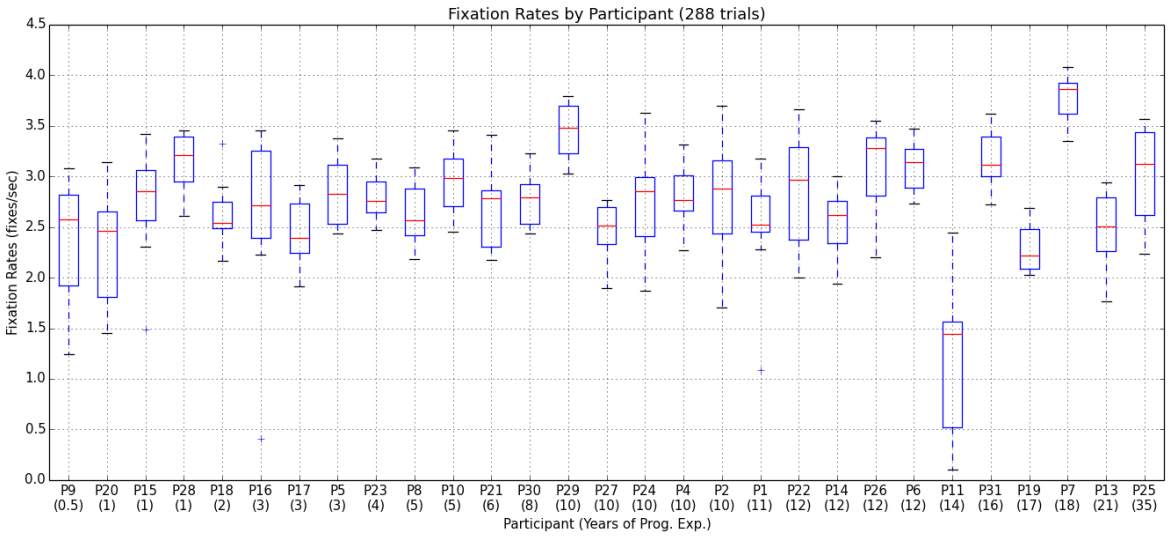


Figure 21: *Mean fixation rate for each trial, grouped by participant. Participants are ordered from least to most programming experience.*

between different tasks. When reading natural language text – e.g., a paragraph of English – we might expect most saccades to be *short* and to the *right*, with a handful of *long* saccades to the *left* (when jumping to the next line). The right-most polar plot of Figure 22 shows precisely this: saccade angles and distances collected from a reading task in which 12 participants read and answered questions about a paragraph of English text [7]. Each dot represents a single saccade's angle (0°to the right) and length (distance from center, proportional to Euclidean distance of longest saccade).

If we examine the saccade angles between fixations over code, we find large differences with the natural language task. The left-most plot in Figure 22 contains saccades from all participants and trials in our experiment that *occurred over the program text* (i.e., excluding the output box). Angles are significantly more spread out in the vertical direction (90°and 270°), and there appear to be equal numbers of long right and left saccades. While code is often considered text when modeling program comprehension [15], it is clearly read differently than English! This aggregate view may be slightly misleading, however. If we focus on a single program base – `counting` in the middle plot of the same figure – saccades are much less vertical. This makes sense given that the `counting` programs are only 3 and 5 lines long. The English text was 4 lines long, begging the question: would a multi-paragraph natural language task produce a saccade plot like the left-most or right-most plot of Figure 22? We will leave this question for future work.

5.1.2 Code Element Fixations

Using the Pygments lexer, we divided code elements into five categories: identifiers, operators, literals, keywords, and conditions. Identifiers are simply the names of functions and variables. Literals are lists, strings, numbers, and boolean values like `True` and `False`. Operators were either mathematical (e.g., `+`, `*`) and set operations like `in`. Conditions involved numerical comparisons, such as `<` and `>`. Across all participants and trials, Figure 23 shows the total fixation durations for each code element category, both

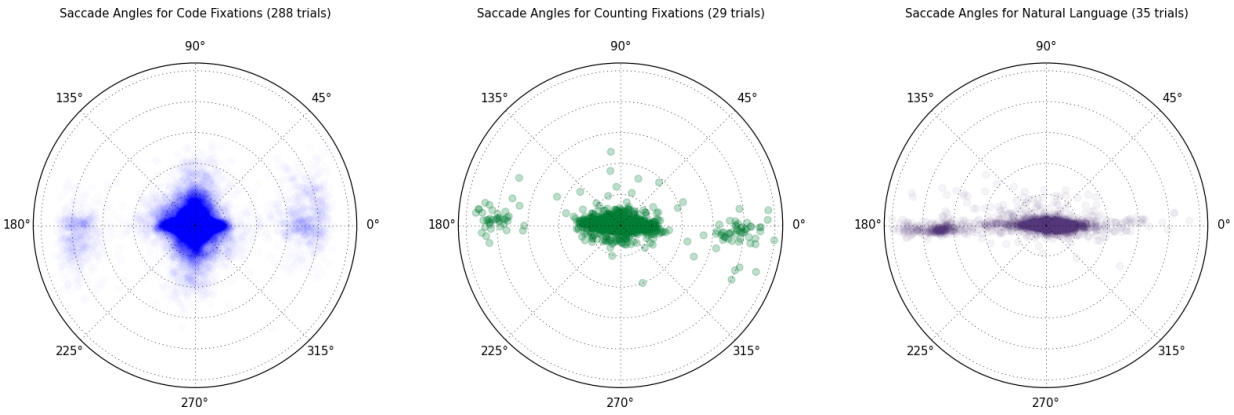


Figure 22: *Saccade angles for all code fixations (left), from just the* `counting` *programs (center), and from fixations in a natural language reading experiment (right). Distance from center is proportional to the longest saccade.*

raw totals (left) and normalized by the number of characters in each category (right). We can see that, per character, keywords receive the least amount of fixation time, followed by identifiers and literals. Operators and conditions received the most normalized fixation time – a result we should expect given the prominence of simple calculations and conditional (`if`) statements in our programs. Literals, such as lists and strings, received most of the total fixation duration in individual programs (see Section 5.2), but their large textual size puts them in the middle of the normalized plot. Our results here are mostly in line with other program comprehension experiments, though every experiment categorizes code elements slightly differently (see Discussion for details).

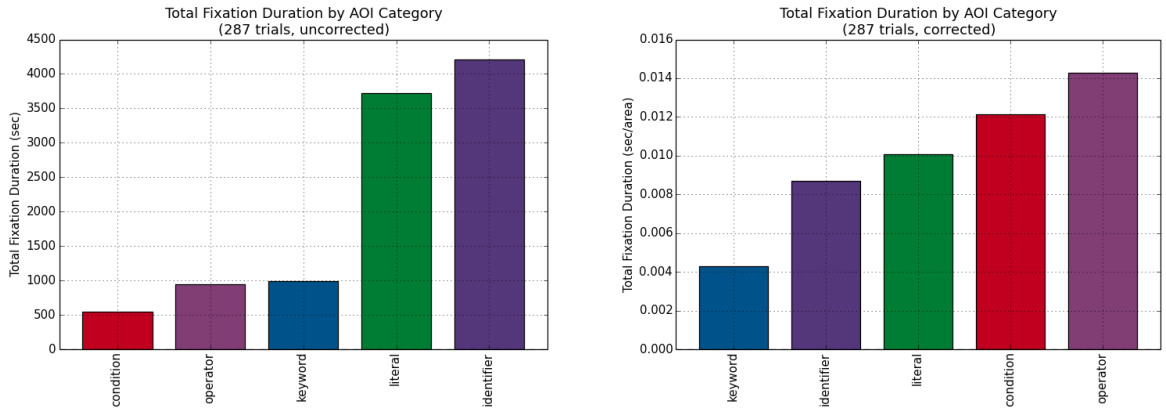


Figure 23: *Total fixation durations by code element category.* ***Left****: raw totals by category.* ***Right****: totals divided by areas of category elements across all programs.*

5.1.3 Scanpath Comparisons

In an attempt to identify common strategies, we compared the scanpaths of participants viewing the same program using the normalized Levenshtein (string-edit) distance [30]. Two kinds of scanpaths were

computed: (1) between whitespace-separated blocks of code, and (2) between individual lines of code. The output box was included in both kinds of scanpaths, and repeated visits to the same area of interest were removed. For each program, clusters of scanpaths with low edit distances (e.g., below 0.25) would represent common ways of reading and evaluating the program.

We were surprised to find large differences between participant scanpaths. On average, we observed a normalized edit distance of 0.52 for block/output scanpaths and 0.65 for for line/output scanpaths (Figure 24). These values indicate that participant looking behaviors were highly individualized – i.e., a given pair of scanpaths from the same program tended to differ more than 50%. At the level of individual program versions, the mean and median edit distances were always greater than 0.25. The closest was `funcall space` with a median of 0.26.

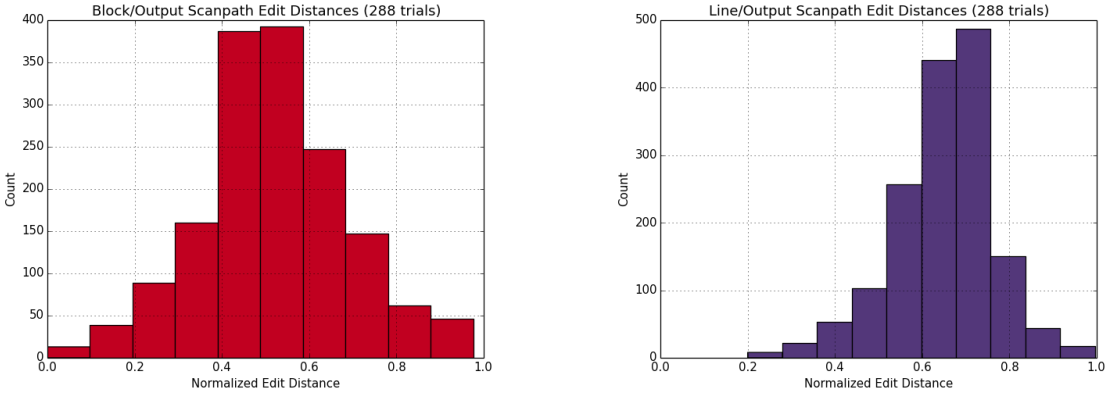


Figure 24: *Normalized edit distances between participant scanpath pairs.* ***Left****: code block and output box scanpaths.* ***Right****: code line and output box scanpaths.*

There are two likely reasons for the large difference between participant scanpaths. First, our task was fairly free-form – participants were not constrained to read or respond in any specific order. Second, there were no time constraints on individual trials, producing scanpaths with variable lengths. The edit distance metric we use is sensitive to both of these complications. Variations of the edit distance exist, with different costs for insertions, deletions, replacements, and even transpositions (e.g., `ba` to `ab`). Methods for normalizing scanpaths also exist, such as removing two character repeats (e.g., `abab` becomes `ab`) to avoid capturing refixations and simply clipping all scanpaths to the same length. Lastly, entirely different comparison algorithms can be used – e.g., ScanMatch [12], scanpath entropy [20]. Because of the large space of possibilities here, we will leave more sophisticated scanpath clustering for future work.

5.2 By Program

In this section, we break results down by individual program base and version. We begin by examining participant performance, graded by how closely their responses approximated the programs true printed output. A *perfect* grade was given if there was a match, character for character. Allowing some room for error, a *correct* grade was given to a response that matched the true output, sans some whitespace and formatting characters (e.g., commas, brackets). Figure 25 shows the proportion of correct (green) and incorrect (red)

responses for each program version. It's immediately apparent that `scope`, `counting`, and `between` were the most difficult for participants. Because there were only 29 participants in our experiment, however, a statistical argument for performance differences between program versions is not feasible. Our larger study with Mechanical Turk participants aims to answer these kinds of questions.

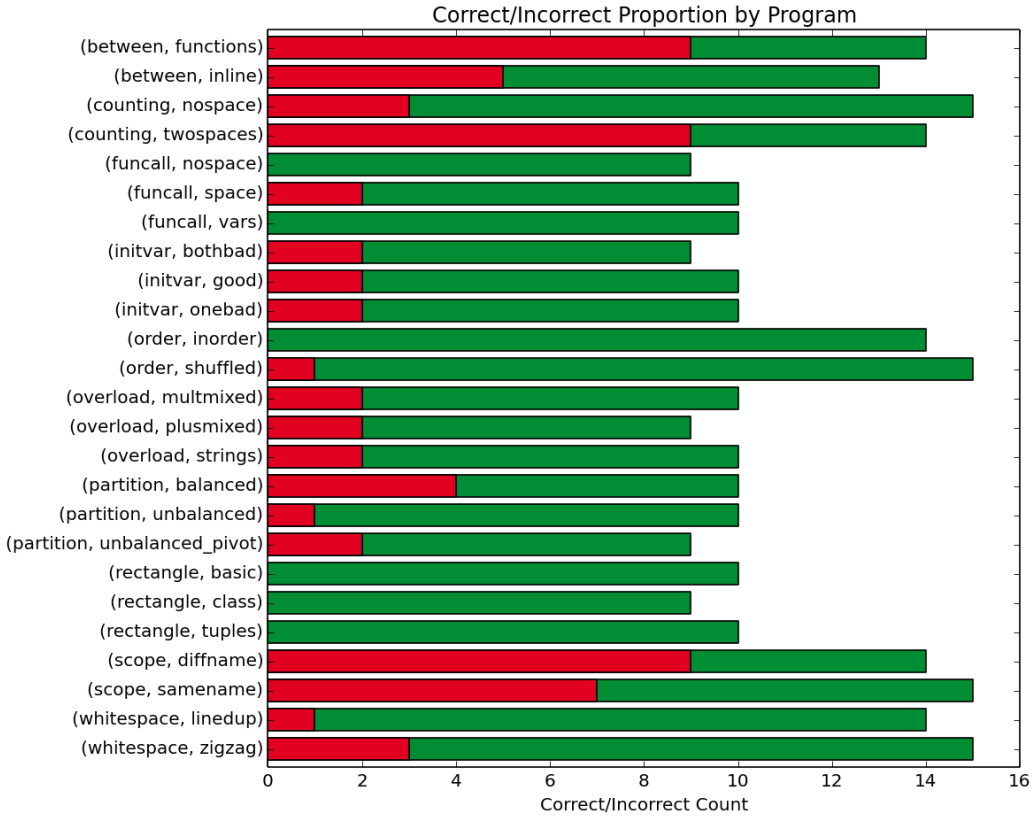


Figure 25: *Correct (green)/incorrect (red) trial proportions and counts by program version.*

5.2.1 between

The `between` programs tested the effects of pulled-out versus inline functionality (i.e., putting code into functions versus repeating it). In both versions, two lists are filtered and printed, and then the common elements in the original lists are printed. We saw several shared looking behaviors between both versions. For example, participants tended to give the majority of their fixation duration to the `x` and `y` lists, and slightly less time to the first comparison in the filtering operation.

The `functions` version contained two functions (between and common), corresponding to the filtering and intersection operations performed on the two lists. Figure 26 shows where participants spent their time when looking at the code. As we might expect, most of the time was spent looking at the two lists (lines 15 and 19). Interestingly, we can see that slightly less time was spent on the second list (line 19). This pattern recurs throughout many of the programs, suggesting that participants are *faster* at evaluating the

second instance of the `between` function. While this might be expected for a function call, the same pattern is observed in the `inline` version (Figure 29), where the code is duplicated.

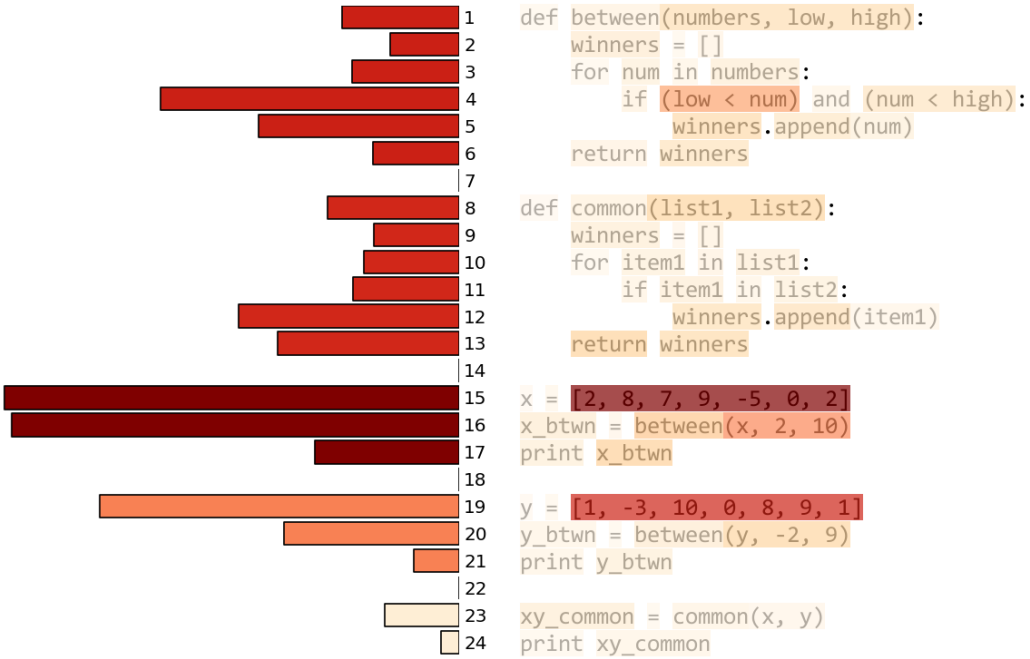


Figure 26: *Total fixation duration for `between` `functions` broken down by screen region, line, and token. Dark red regions have the highest concentration of fixation time (all trials).*

Inspecting the transition matrix for `functions`, we can observe some expected behavior. The highest probability transitions *from* the output box are to lines 15-19 – where the two lists are located. Transitions *to* the output box involve these lines, but also include the `common` function call on line 23. Notable transition probabilities also occur between lines 15 and 19, something we would expect if participants were comparing both lists to find the common elements. To see the actual process of evaluation, though, we need to move beyond the aggregate transition matrix and focus on the dynamic behavior within single trials.

Using a timeline of fixations on each line of code (and the output box), we can observe distinct phases of evaluation at the level of an individual trial. Figure 28 shows a trial timeline with five different sections highlighted in *blue* from left to right, representing what we believe to be the participant (1) reading the `between` function, (2) reading the `common` function, (3) filtering the first list, (4) filtering the second list, and (5) finding the common elements between lists. Additional evidence comes from this participant's intermediary responses (highlighted in yellow and listed in Table 4). These responses line up nicely with our hypothesized order of evaluation by coming at the ends of the last three sections.

The `inline` version did not contain any function definitions, and instead repeated code for the filtering and intersection operations. Figure 29 shows the relative amount of time spent on each line of this version. Like `functions`, less time is spent on the second instance of the filtering operation (lines 8-13), suggesting that participants recognized the repeated pattern. Unlike `functions`, however, relatively little time was spent on the intersection operation (lines 15-19) compared to the `common` function. Our larger study with participants from Amazon's Mechanical Turk found a slight, though significant, increase in correct responses

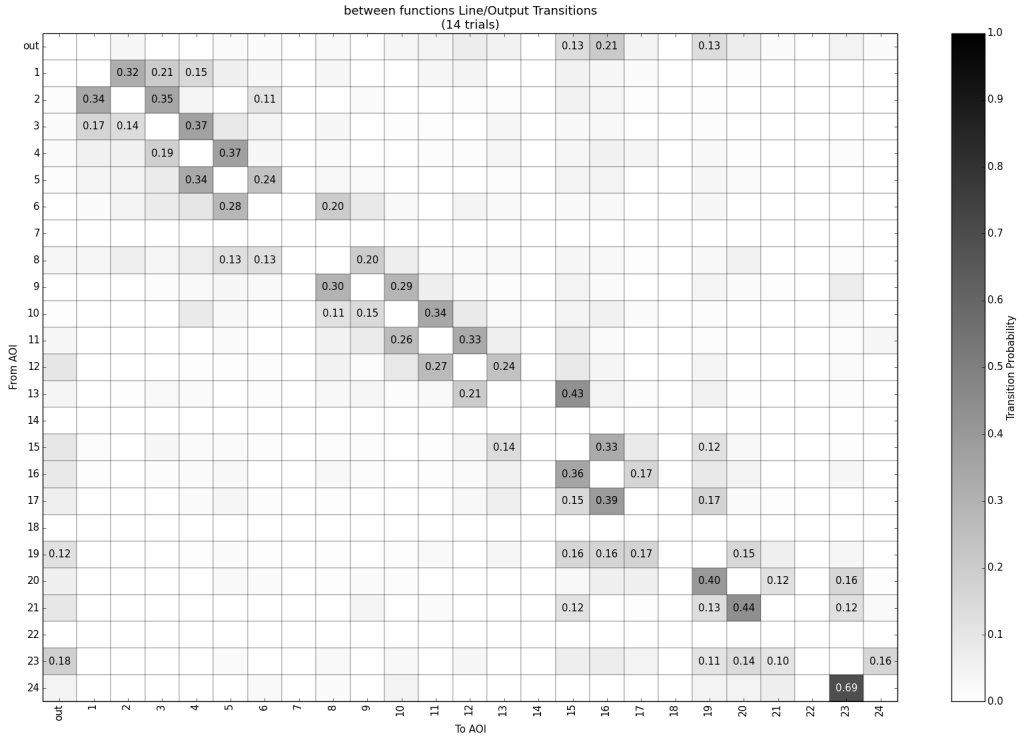


Figure 27: *Transition matrix for* `between functions` *(all participants). Probabilities below 0.1 are not annotated.*

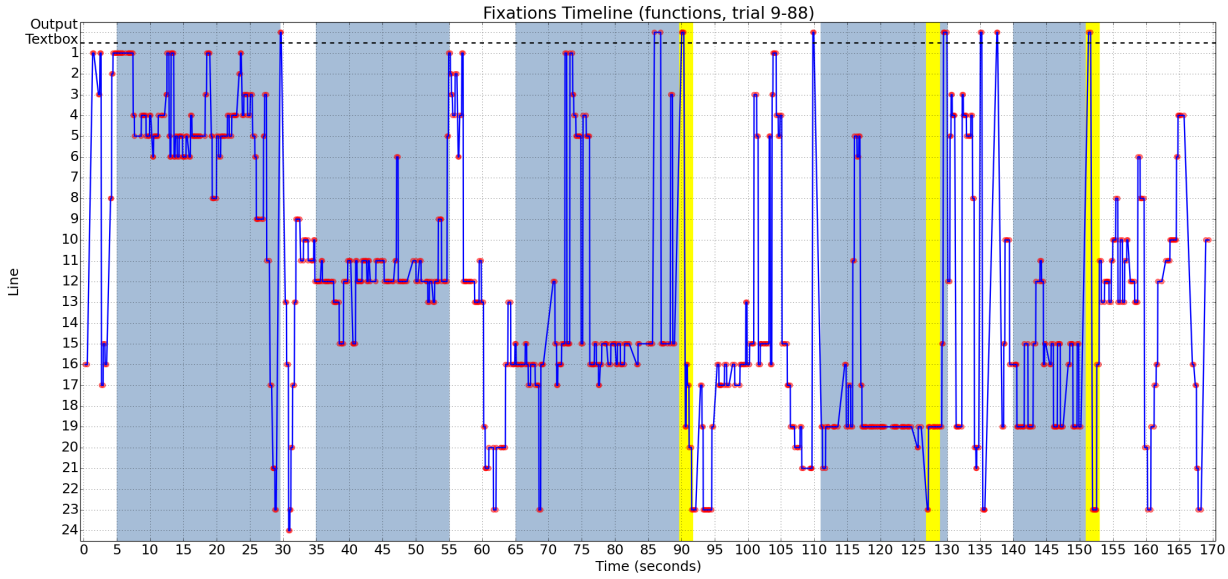


Figure 28: *Timeline of fixations by line for a single* `between functions` *trial. Regions in blue correspond to expected steps in evaluation. Yellow regions match response times from Table 4.*

time_ms	response
90689	[8,7,9]
127881	[8,7,9]•[1,0,8,1]
151920	[8,7,9]•[1,0,8,1]•[8,9,0]

Table 4: *Intermediary responses for trial 9-88 (`between functions`). The • character represents newlines.*

for `inline` versus `functions`.

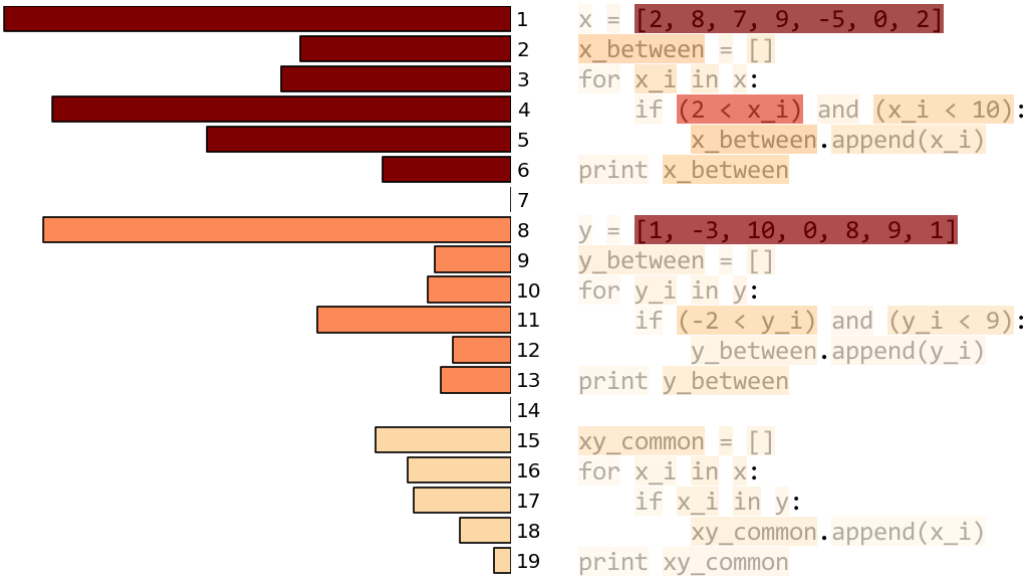


Figure 29: *Total fixation duration for `between inline` broken down by screen region, line, and token. Dark red regions have the highest concentration of fixation time (all trials).*

Based on the eye movement data summary here, it seems reasonable that the intersection operation in `inline` was more recognizable than in `functions`. When inspecting response errors in the larger study, however, we found that participants were no more likely to make a mistake on the final line of output (`[8, 9, 0]`) in *either* version. This output line – corresponding to the intersection operation – was five times more likely to contain an error than the first output line, and over twice as likely than the second. A better hypothesis might be that participants simply mistook the intersection operation for something else. Over 40% of participants who answered incorrectly (in both versions) provided an `[8]` as the last output line. This mistake is consistent with assuming that the intersection operation occurs between `x_between` and `y_between` rather than `x` and `y`. Perhaps, then, participants spent less time on the intersection operation because they had (sometimes incorrect) assumptions about what it was "supposed" to do.

5.2.2 counting

The `counting` programs tested the effects of whitespace on the grouping of statements in a `for` loop. Both versions of `counting` did the same thing: print "The count is i" and "Done counting" for $i \in [1, 2, 3, 4]$. While the `nospace` version had the `for` loop declaration and the two `print` statements in the body on consecutive

lines, the `twospaces` version added two blank lines between the first and second `print` statement. Because Python is sensitive to indentation, this did not change the semantics of the program (i.e., both `print` statements still belonged to the `for` loop).

The extra whitespace in the `twospaces` version clearly had an effect: only 36% of participants provided a correct answer (as opposed to 80% correct in the `nospace` version). Proportionally, participants spent more time on the last print statement in the `nospace` version (Figure 30), likely because they were reading it during each evaluation of the loop body. This hypothesis is supported by inspecting the transition matrices for both `counting` versions (Figure 31). Participants were almost twice as likely to fixate the "Done counting" line after the first `print` statement on line 2 in the `nospace` version than in the `twospaces` version.

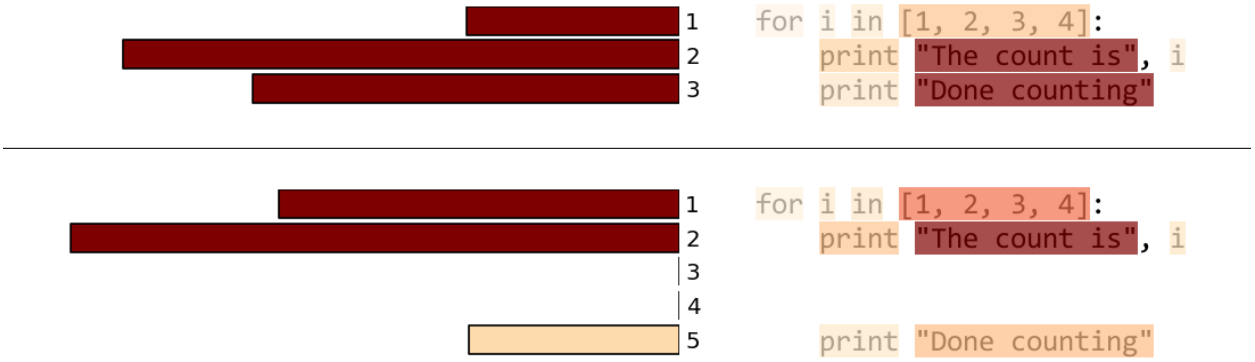


Figure 30: *Total fixation durations by line for both versions of the* `counting` *program (all participants). Top:* `nospace`*, bottom:* `twospaces`*.*

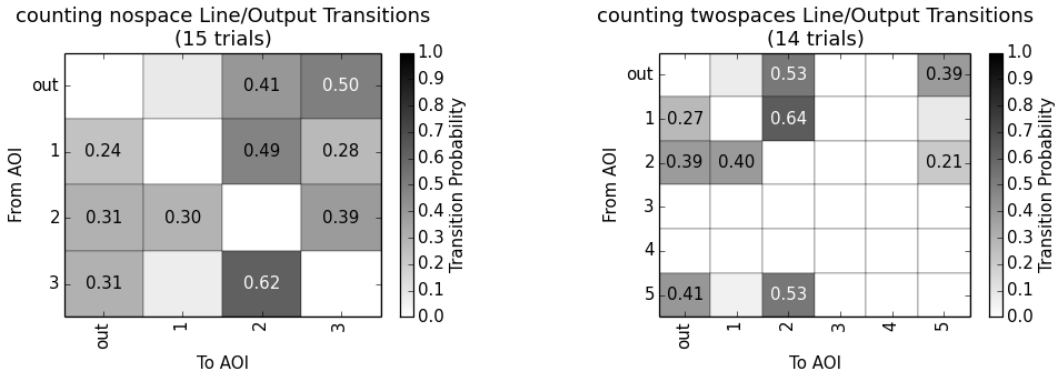


Figure 31: *Transition matrices for* `counting` *programs (all participants). Probabilities below 0.1 are not annotated.*

We can even see a difference in the `twospaces` trials alone if we split them out by correct and incorrect responses. The transition matrices reveal an interesting difference: participants that provided a correct response were about twice as likely to fixate on the "Done counting" line after line 2 (Figure 32). Examples of individual trials where this behavior was observed are shown in Figure 33. In the correct trial (top), the participant reads all lines of the program before visiting the output box to start typing their response. The participant in the incorrect trial, however, visits the output box after mainly fixating on line 2, and tends to

fixate the "Done counting" `print` statement (line 5) in isolation.

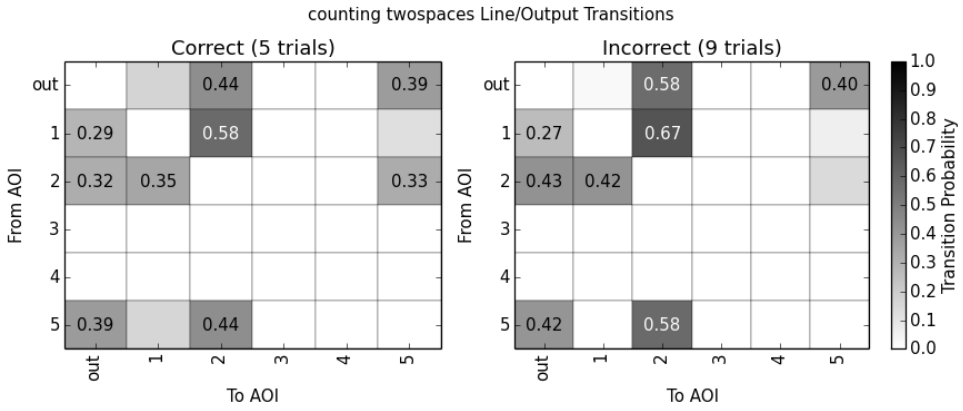


Figure 32: *Transition probabilities between lines and the output box for the* `twospaces counting` *program.* ***Left****: trials with correct responses.* ***Right****: trials with incorrect responses.*

The data from `counting` demonstrate that reading behavior and program interpretation are strongly linked. We are not suggesting, however, that simply failing to spend sufficient time looking at the "Done counting" line *causes* a participant to produce an incorrect response. Rather, this is likely a symptom of an underlying error when spatially grouping the `print` statements. Would the same pattern of errors occur if the entire program was embedded in a larger `for` loop or function? Further work is needed to determine which spatial cues programmers use to group code elements, and how `twospaces` is violating the assumptions behind those cues.

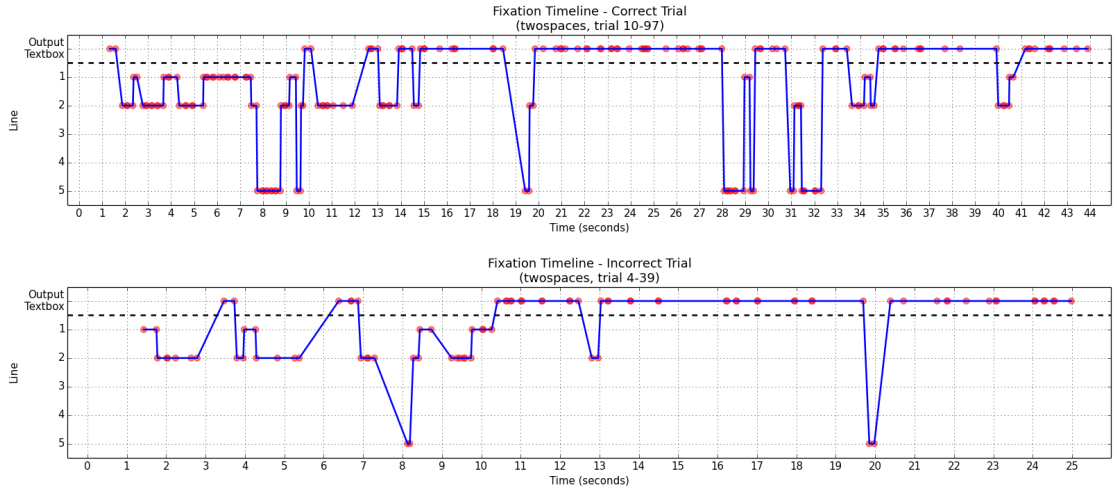


Figure 33: *Fixation timelines for two* `twospaces` *trials. The top trial resulted in a correct response, while the bottom trial did not.*

5.2.3 funcall

The `funcall` programs each performed a compound calculation using a single, simple function: $f(x) = x + 4$. The calculation, $f(1) \times f(0) \times f(-1)$, either had no whitespace between terms (`nospace` version), a single space between all tokens (`space`), or had each call to $f(x)$ bound to a variable before completing the calculation (`vars`). We expected more whitespace to facilitate faster trials and more correct responses, especially in the `vars` versions where the calculation is broken out into multiple, named steps (i.e., `x`, `y`, `z`).

Trial time and response correctness across all versions was effectively the same, violating our performance expectations. In terms of fixation time, the results are also largely the same across versions: the arguments provided to f and the + operator are focal points (Figure 34). The `return` keyword on line 2 is fixated more often than we would expect – keywords received the smallest fixation duration per area across all programs in our experiment (Figure 23). Looking at the transition matrices for each version (Figure 35), we can see that line 2 is a common destination. This is especially true from lines with f calls (line 4 in `nospace` and `space`, lines 4-6 in `vars`). We suspect that the `return` keyword served as a beacon for visual jumps back to the addition on line 2, thus increasing its share of relative fixation time.

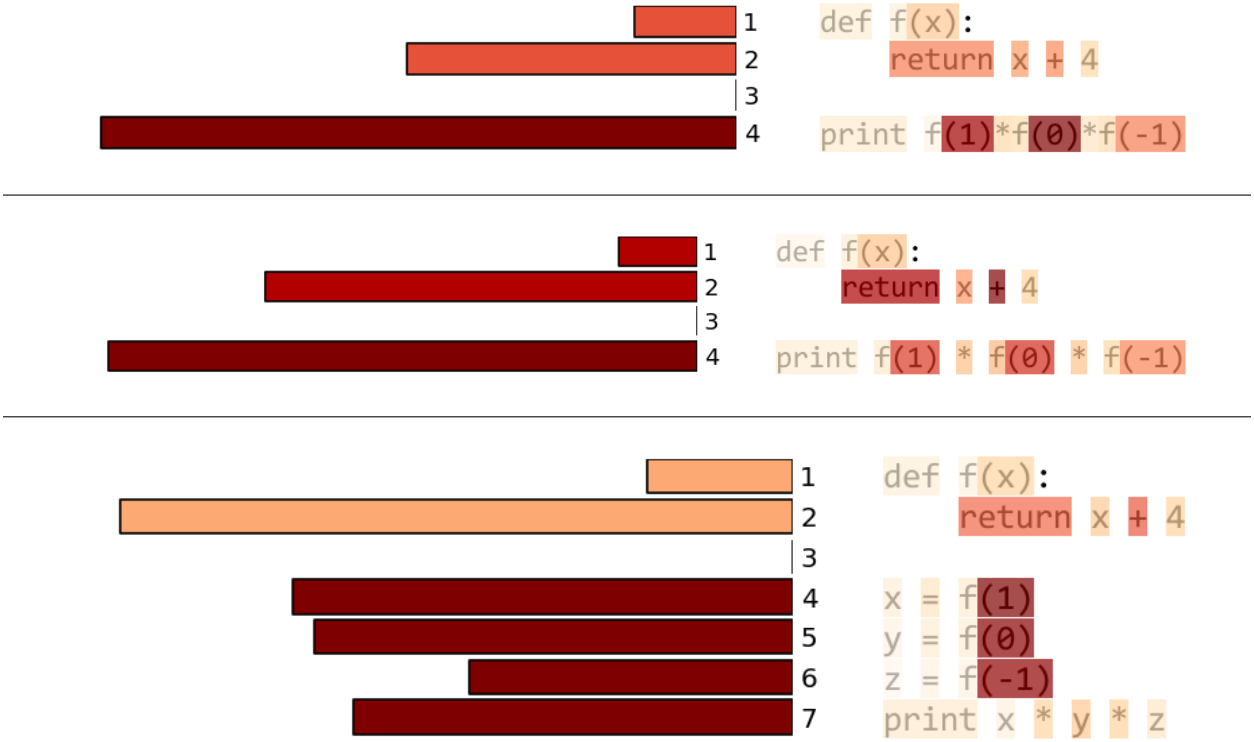


Figure 34: *Total fixation durations by line for all three versions of the* `funcall` *program (all participants). Top:* `nospace`*, middle:* `space`*, bottom:* `vars`*.*

The transition matrix for the `vars` version reveals steadily decreasing jump probabilities back to line 2 for each call of f (lines 4-6). This suggests that participants needed to look up the definition of f less and less as they performed the calculation. A good example of this behavior is shown in Figure 36. The participant fixates lines 1 and 2 three times in the first five seconds, and twice in the next five seconds. By the end of

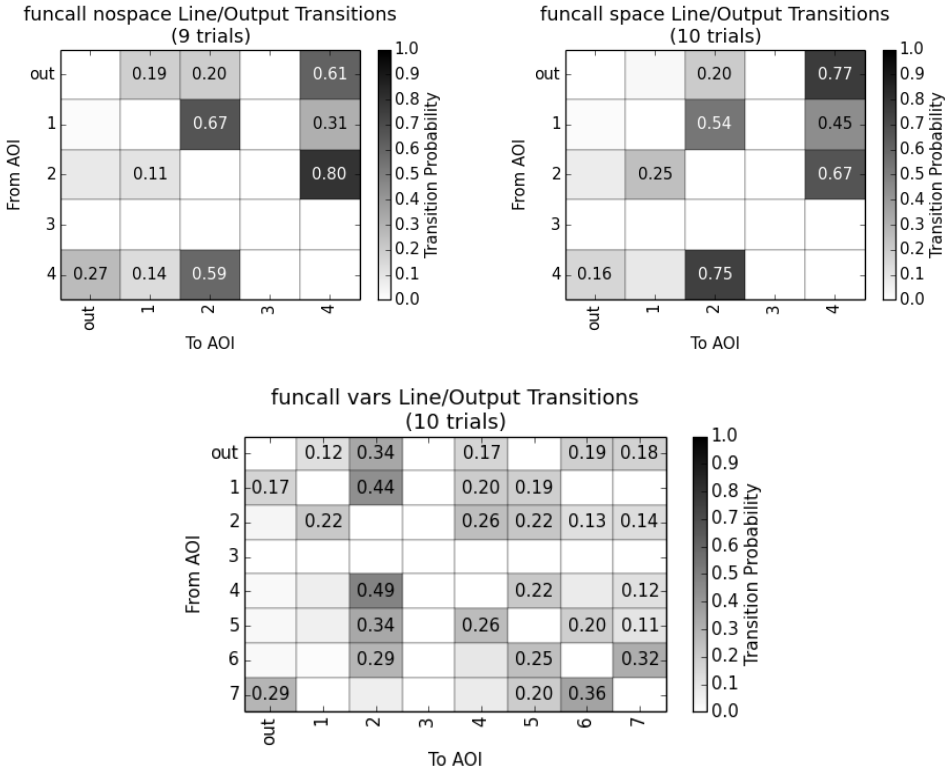


Figure 35: *Transition matrices for* `funcall` *programs (all participants). Probabilities below 0.1 are not annotated.*

the trial, only the output box is being fixated, presumably as the participant calculates the printed product. Looking at the participant's keystrokes partially confirms this hypothesis – an intermediary result of 12×5 is typed first before being replaced by a 60 at about 26.8 seconds.

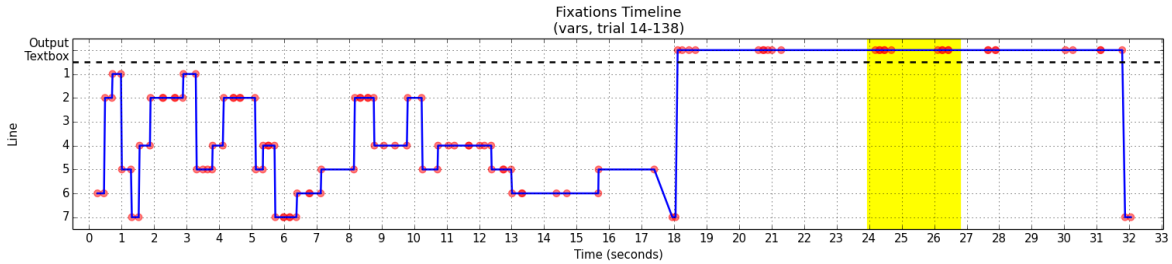


Figure 36: *Fixation timeline for* `vars` *trial. The highlighted region (yellow) goes from the participant's first keystroke to the last.*

Let's dig just a little deeper. Looking at the average fixation duration over a rolling window across the same trial, we can see two spikes around 17 and 27 seconds (Figure 37). Given the rolling window delay, these events likely correspond to the initial computation of the three additions ($1 + 4$, $0 + 4$, $-1 + 4$), and the later computation of the final product ($5 \times 4 \times 3$). Similar correspondence between average fixation duration over a rolling window and mental computation was found in the `initvar` programs (Section 5.2.4).

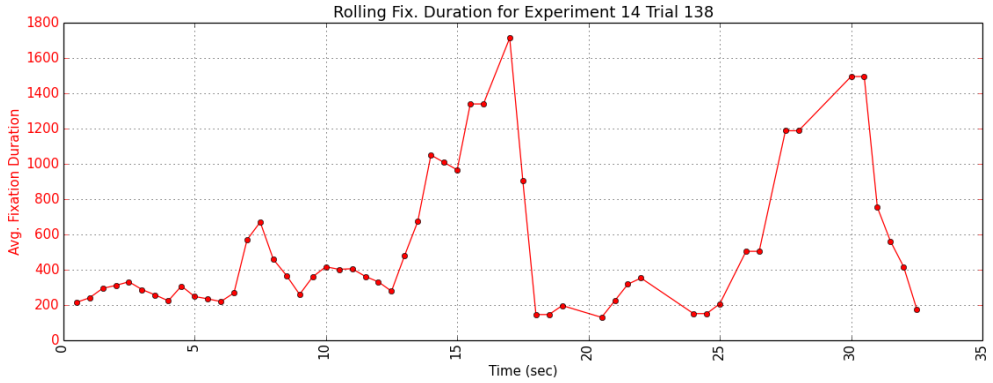


Figure 37: *Average fixation duration over a 1 second rolling window (1/2 second step).*

5.2.4 initvar

The `initvar` programs each contained two accumulation loops: one performing a product, and the other performing a summation. In the `good` version, both loops were intended to meet expectations; the product loop had an initial value of 1, and the summation loop had an initial value of 0. The `onebad` version, however, started the summation loop at 1 (an off-by-one error). The `bothbad` version contained the same error, and also started the product loop at 0 (making the final product 0).

The fixation durations and transition matrices for all versions were fairly similar, except for one major difference: participants in the `bothbad` version spent much less time on the list in the first `for` loop (Figure 38).

This provides evidence for our suspicions that participants would "short-circuit" the product calculation by noting that $a = 0$, and thus any product with a would also be 0. Performance-wise, this resulted in a significant increase in response proportion – i.e., a higher proportion of `bothbad` trials were spent responding.

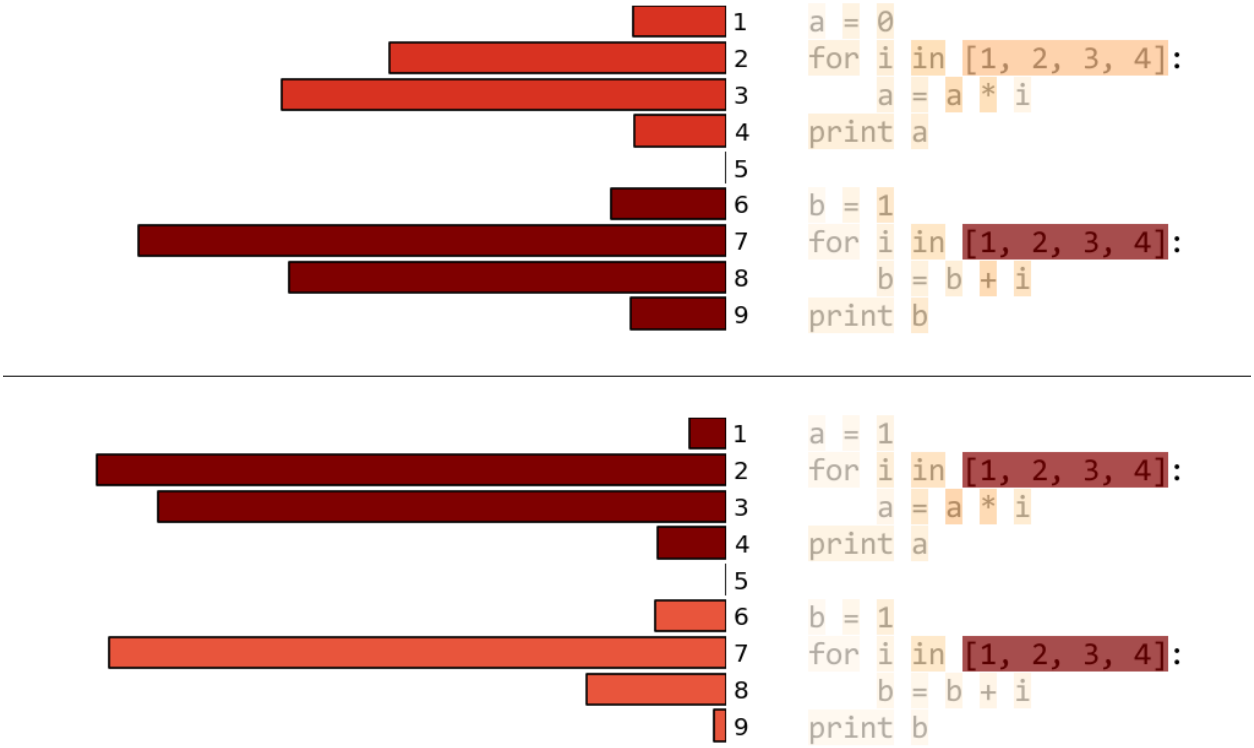


Figure 38: *Total fixation durations by line for two versions of the `initvar` program (all participants). Top: `bothbad`, bottom: `onebad`.*

We expected the `bothbad` transition matrix (Figure 39) to have a large transition probability from lines 2 or 3 to the output box, representing a short-circuited calculation (and entry of a "0" by the participant). We did not observe this in the aggregate across all `bothbad` trials, but the behavior is evident in some of the individual trials. For example, Figure 40 shows the timeline of trial 59 (participant 6). The red line shows when the participant first typed a "0" in the output box, which occurs immediately after looking at the first loop. We can be confident this represents a short-circuited calculation because this participant's keystrokes (Table 5) contained intermediary results for the remaining calculation.

Another source of evidence for a short-circuited calculation in the same trial comes from the average fixation duration over a rolling window (Figure 41). The largest spikes occur right around when the participant has finished typing the operands for the final calculation (60-70 sec), and **not** when looking at the first calculation. Interestingly, a small spike occurs around the 50 second mark – right when the participant corrects an error in their response. Average fixation duration may indicate intermediary errors as well as mental calculations.

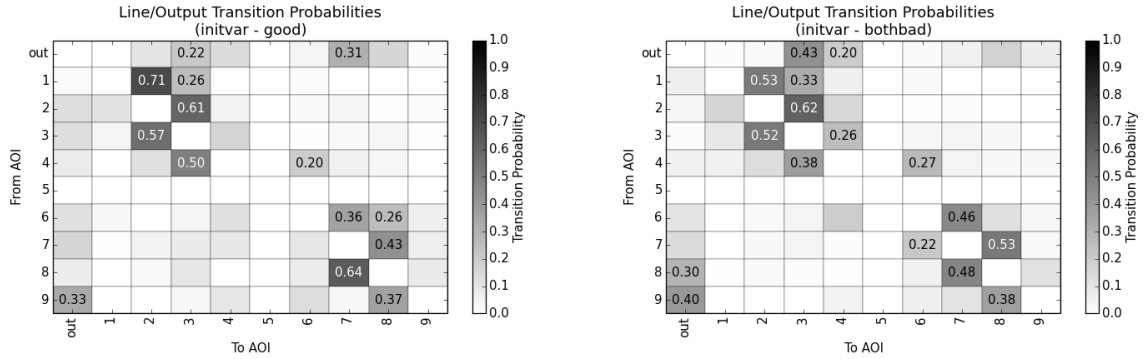


Figure 39: *Transition matrix for* `good` *and* `bothbad` *trials. Probabilities below 0.1 are not annotated.*

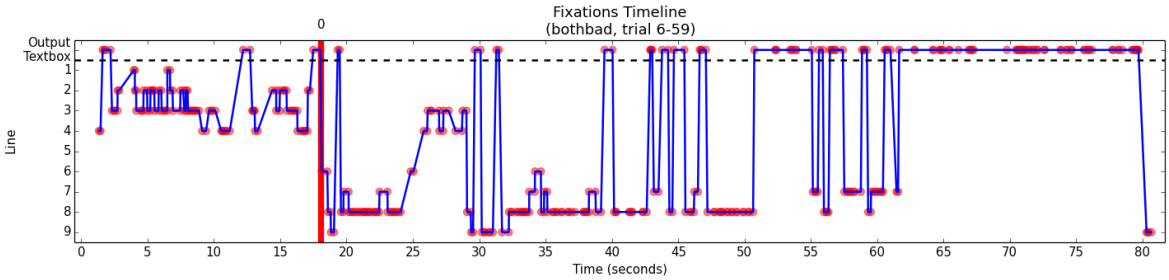


Figure 40: *Fixation timeline for* `bothbad` *trial. The red line indicates when a "0" response was given.*

time_ms	response
17848	0
19294	0●
40125	0●1
43821	0●1+
45245	0●1+1
46765	0●1+1+
50837	0●1+1+3
51853	0●1+1+2
52205	0●1+1+2+
54669	0●1+1+2+3
55182	0●1+1+2+3+
55645	0●1+1+2+3+4
64188	0●1+1+2+3+4
64332	0●1+1+2+3+4 =
64836	0●1+1+2+3+4 =
70900	0●1+1+2+3+4 = 1
71037	0●1+1+2+3+4 = 11
81000	0●11

Table 5: *Time series of responses for trial 59, participant 6 (*`initvar bothbad`*). A ● represents a new line.*

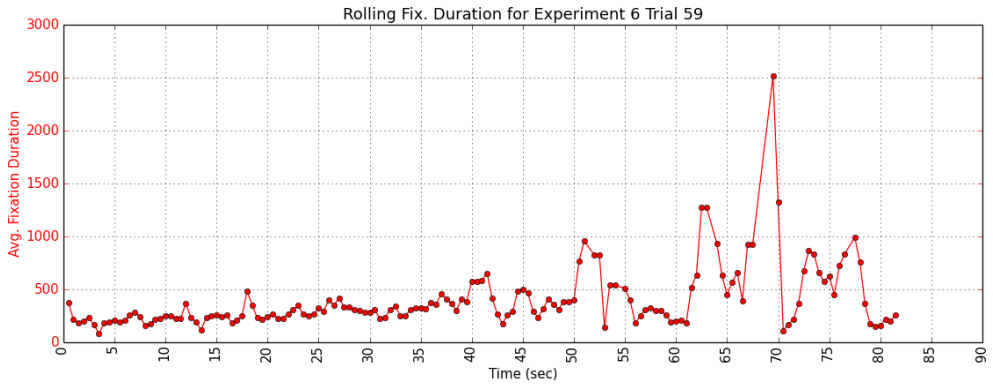


Figure 41: *Average fixation duration over a 1 second rolling window (1/2 second step).*

5.2.5 order

The `order` programs contained three functions, $f(x)$, $g(x)$, and $h(x)$. f and g added 4 and doubled x respectively, and h computed $f(x) + g(x)$. In the main body, all three functions were called on $x = 1$ in the same order: f, g, h. The order in which the three functions were defined was either in the called order (`inorder`), or in a slightly different order (`shuffled`) – h, f, g.

We found that participants in the `shuffled` trials took slightly longer to provide responses, presumably because there were ordering expectations for the definitions of f, g, and h. The visualizations of where participants spent their time in both versions are very similar, with the exception that a bit more time was spent in `shuffled` on the `return` keywords inside each function (Figure 42). Like the `funcall` programs, we suspect that these code elements served as beacons for quickly locating the function bodies. The implicit ordering of functions in the `inorder` version may have lessened the need for such beacons. This hypothesis is not quantifiable, however, without further experimentation.

Surprisingly, we did not see differences between versions in the transition matrices, or in metrics related to visual search efficiency, such as normalized scanpath length, fixation rate, spatial density, and average saccade length. Given the observed difference in trial times in our larger study (`inorder` participants were slightly faster), we expect that there exists *some* quantifiable method for distinguishing between participants' eye movements in both versions. We may need more participants, however, to locate a distinguishing feature.

Looking at the timelines for individual trials, similar behaviors can be easily noted (Figure 43). In the `inorder` trial (top), we can see fixations move from lines 1-2 (definition of f) to lines 4-5 (definition of g), and finally to lines 7-8 (definition of h) with occasional jumps back to line 2 (f). Similarly, the `shuffled` trial (bottom) starts on lines 4-5 (f), goes to lines 7-8 (g), and ends with lines 1-2 (h) plus the occasional jump to line 4 (f). The shuffled ordering does not appear to heavily impact visual search or transitions to the output box.

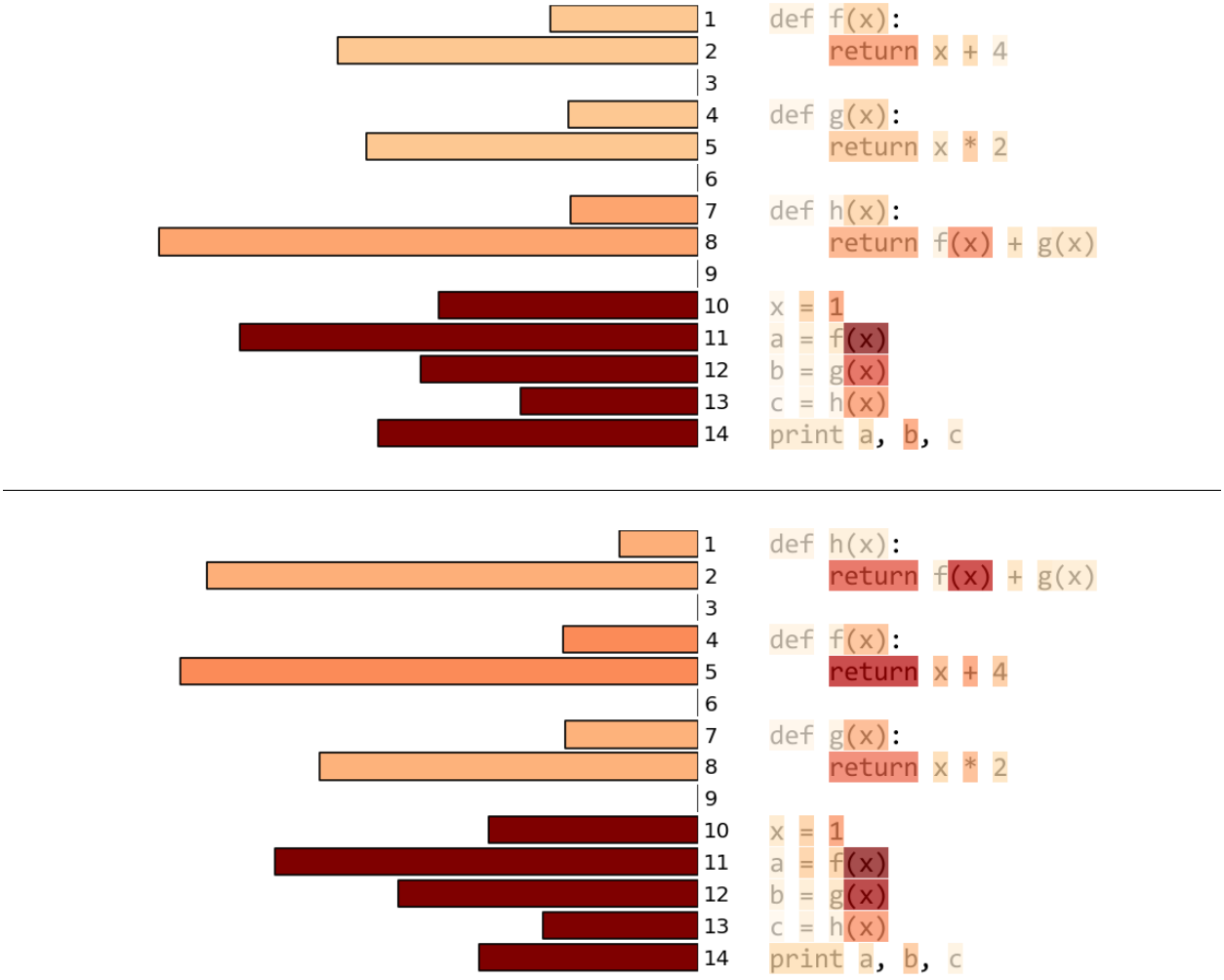


Figure 42: *Total fixation durations by line both versions of the* `order` *program (all participants). Top:* `inorder`*, bottom:* `shuffled`*.*

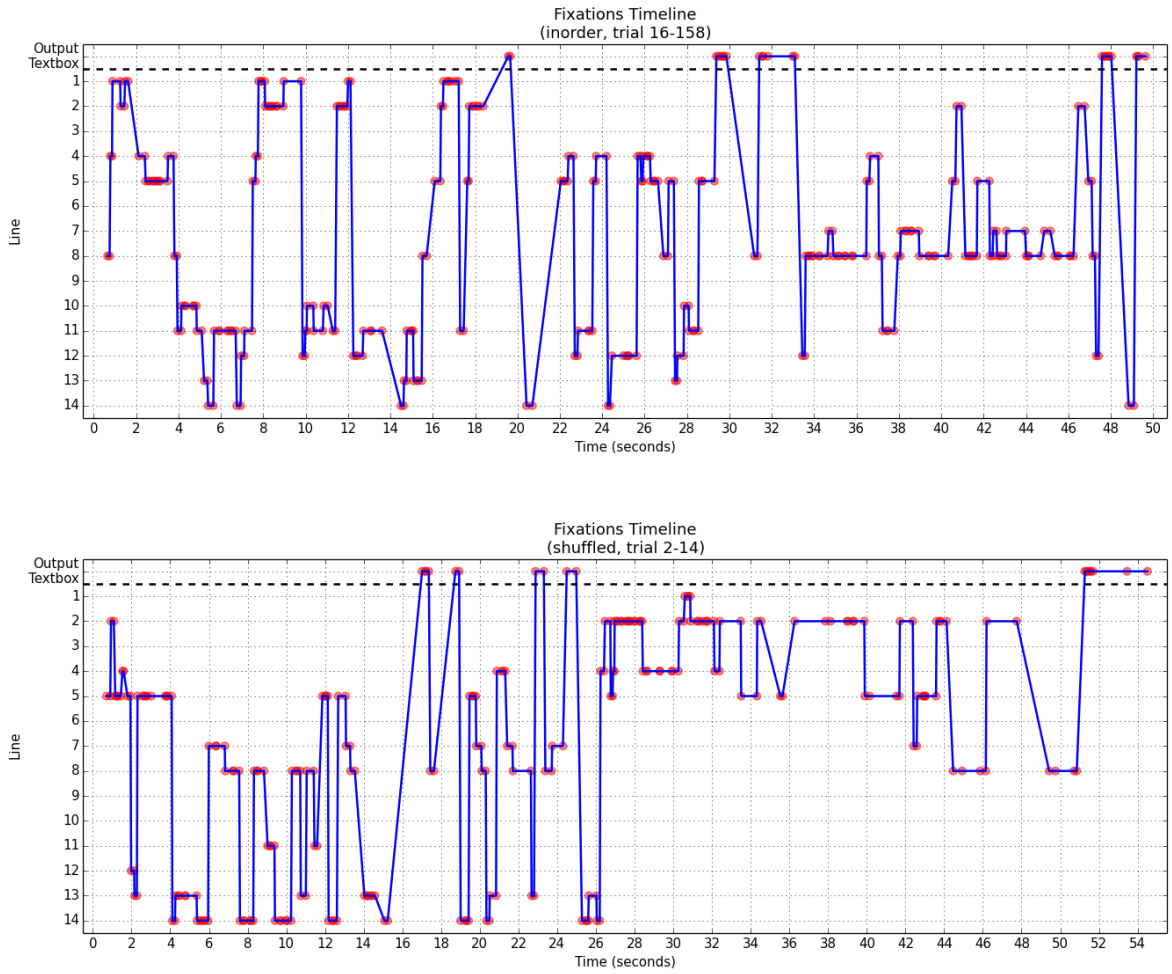


Figure 43: *Timeline of line and output box fixations for two* `order` *trials. Top:* `inorder`*, bottom:* `shuffled`*.*

5.2.6 overload

The `overload` programs tested the effects of operator overloading – i.e., having multiple meanings for the same operator. Each program had three blocks of code, each with two variable assignments followed by an operation with those two variables (and a printing of the result). The final block always assigned a string "5" and string "3" to variables `e` and `f`, and then printed `e + f`. The + operator in Python is *overloaded*, and can either be addition or string concatenation depending on the types of its operands. The preceding two code blocks either had exclusively multiplications (`multmixed`), additions (`plusmixed`), or string concatenations (`strings`).

We expected participants to be "surprised" more by the final + operation (string concatenation) in the `plusmixed` version than the others due to being primed with the mathematical sense of the operator. We saw the largest differences between `plusmixed` and `strings`: participants focused much more on lines 9-11 in `plusmixed`, specifically on the string values and + operator (Figure 44). The results for `multmixed` were similar to `plusmixed`, though not quite as distinct from `strings`.

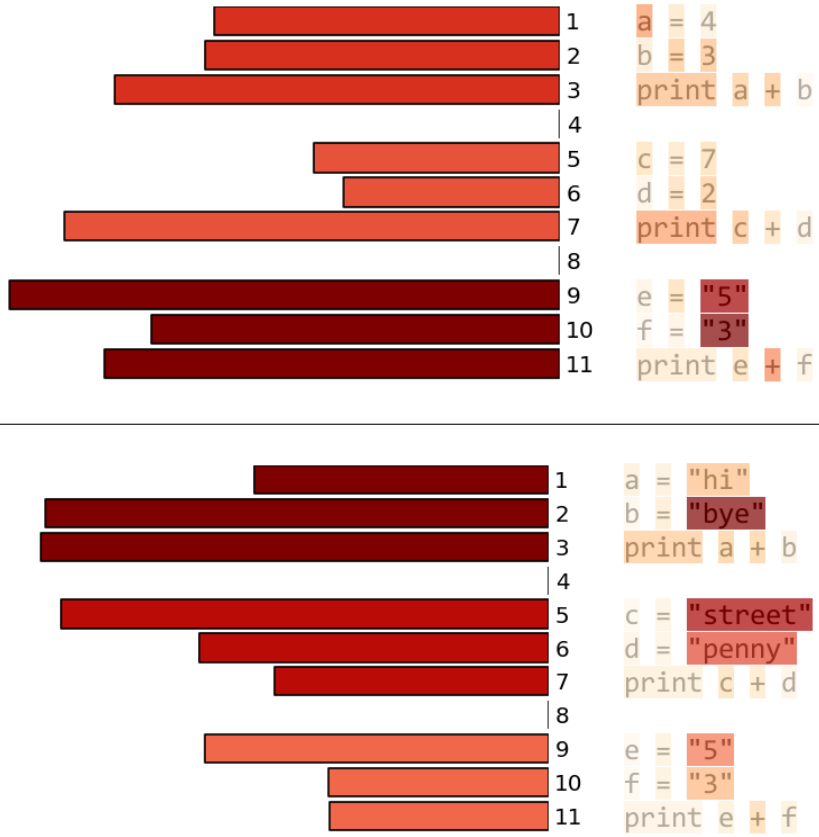


Figure 44: *Total fixation durations by line for two versions of the `overload` program (all participants). Top: `plusmixed`, bottom: `strings`.*

It's unclear whether participants were more "surprised" by the string numerals ("5" and "3") or the + operator. While the transition matrices for `plusmixed` and `strings` were not drastically different (Figure 45), we did see a slightly higher probability of going from the output box to lines 9-11 in the `plusmixed` version

(0.54 versus 0.25). This suggests that participants were more likely to go back and check the string concatenation block of code after responding when it was proceeded by additions. A good example of this behavior is shown in the timeline in Figure 46. After typing the final line of their response ("53"), as indicated by the yellow region, the participant fixates the output box before returning to the final code block. Interestingly, the participant also returns to line 7 (`print c + d`), perhaps to double-check that previous additions were not actually string concatenations. It appears that some priming is going on for the + symbol, and that having multiple senses of an operator in the same small program is enough to affect eye movements.

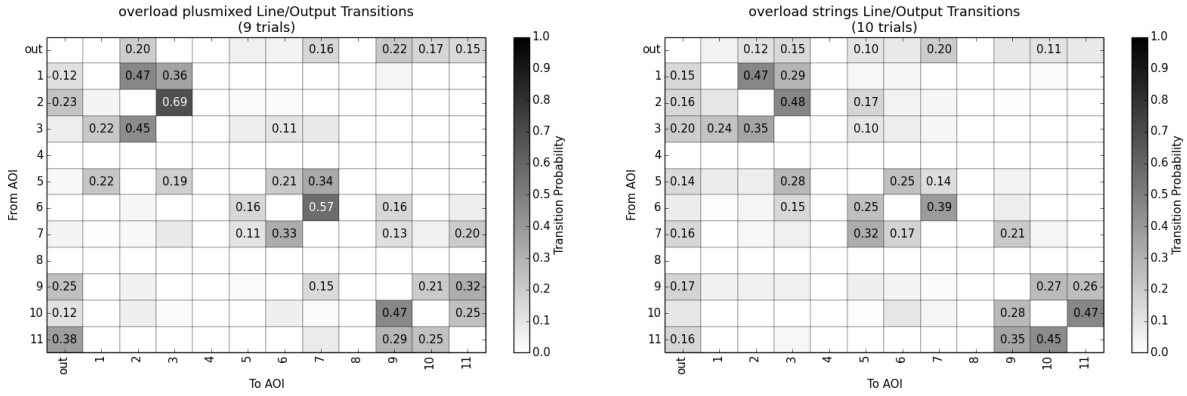


Figure 45: *Transition matrices for* `overload` *programs (all participants). Probabilities below 0.1 are not annotated.*

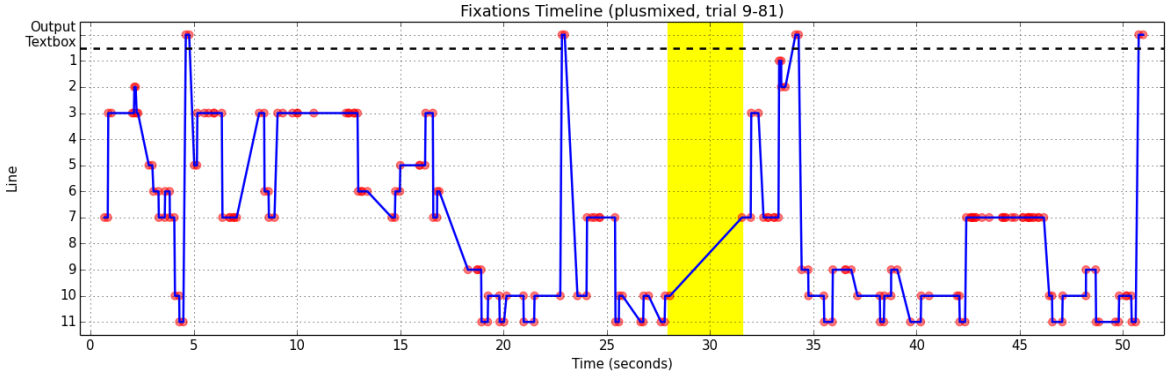


Figure 46: *Fixation timeline for a* `plusmixed` *trial. The yellow region indicates when the participant was typing "53".*

5.2.7 partition

The `partition` programs each iterated through a list of numbers, and printed the number plus a "high" or "low" designation on each line. Numbers less than 3 were *low*, and numbers greater than 3 were *high* (3 itself was skipped). The `balanced` version iterated over $[1, 2, 3, 4, 5]$, producing an equal number of *low* and *high* numbers. In contrast, the `unbalanced` and `unbalanced_pivot` versions iterated over $[1, 2, 3, 4]$, printing two

low and only one *high*. The `unbalanced_pivot` version used variable `pivot = 3` instead of the constant 3 in its `if` statements.

We expected participants to make fewer errors in the `balanced` version because of the symmetric *low* and *high* values. No systematic difference between versions was observed, however. In fact, the most common error was simply forgetting to print the number (`i`) alongside the low/high label. There were no major differences between relative fixation durations either – participants in all three versions spent the majority of their time on the list and two conditionals (Figure 47).

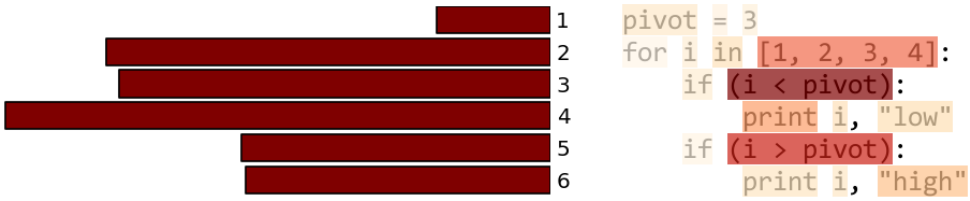


Figure 47: *Total fixation durations by line for the `unbalanced_pivot` version of `partition`. Other versions had similar results.*

The transition matrices for all versions were also very similar, but contained an interesting pattern. Figure 48 shows the transition matrix for the `unbalanced_pivot` trials. Lines 2-4 form a looping core in the matrix, with high probability transitions going to the next and previous lines. From this, we might expect participants to snake their fixations up and down these lines as they evaluate the program. Indeed, this behavior is observable in the fixation timelines – e.g., Figure 49.

This timeline also helps explain the transition probabilities coming out of line 1 (`pivot = 3`). The highlighted portions correspond to the participant's entry of each response line (`1 low`, `2 low`, and `4 high`). We can see several quick jumps up to line 1, one before each entry of the first and second response lines. Line 1 receives few visits, so the high probability transitions from it to lines 2 and 4 are likely due to (1) most participants reading the program in line order, and (2) referencing the variable's value while evaluating the `print` statement on line 4. The latter behavior is referred to as **tracing** in Cant et. al's cognitive model of program comprehension [9] and, along with `chunking`, encompasses much of basic program comprehension.

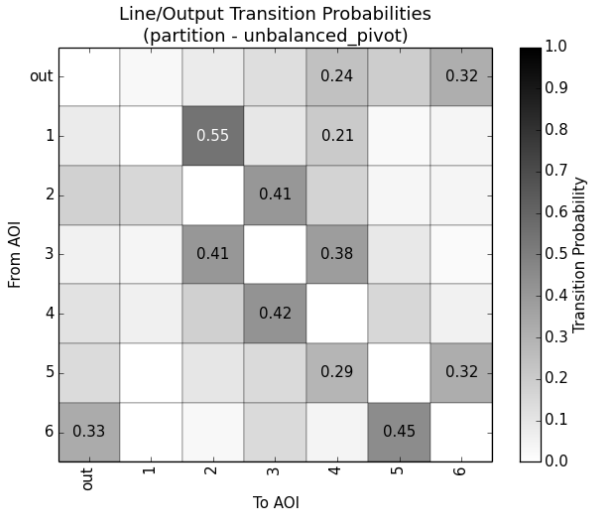


Figure 48: *Transition matrix for all* `unbalanced_pivot` *trials. Probabilities below 0.1 are not annotated.*

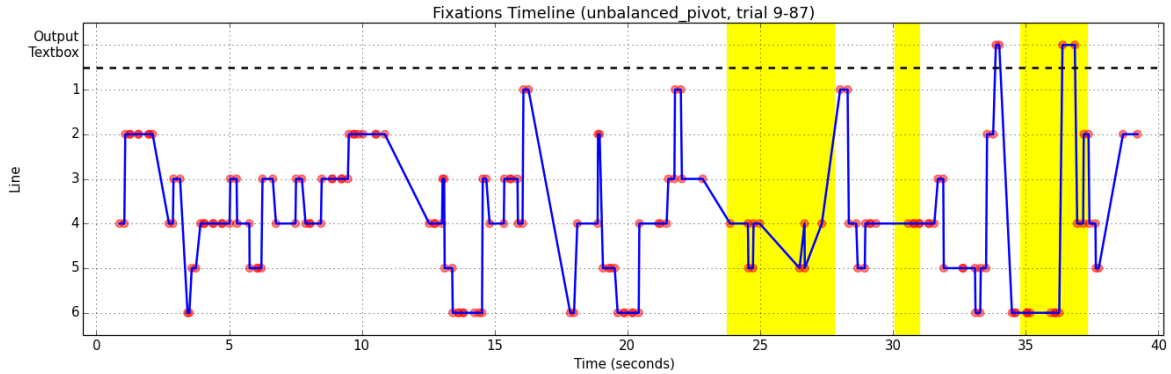


Figure 49: *Fixation timeline for an* `unbalanced_pivot` *trial. The yellow regions correspond to the typing of the first, second, and third response line.*

5.2.8 rectangle

The `rectangle` programs computed the area of two rectangles, represented either as a collection of four `x`/`y` variables (`basic`), a `Rectangle` object (`class`), or a pair of `(x, y)` coordinates (`tuples`). All 29 participants produced a correct response to the `rectangle` programs, and we did not find any significant difference in performance metrics between versions in the present or larger study (with Mechanical Turk participants).

Despite the different forms of the `rectangle` programs, the eye-tracking visualizations reveal strong similarities. In general, participants focused on the ordering of arguments (either to the `area` function or the `Rectangle` class constructor), and on the numeric values for width and height (Figures 50, 51, 52). Interestingly, less time was always spent on the second set of arguments/values, providing evidence for short-term learning of the area operation. Similar behavior was observed for the filtering operation in the `between` programs (Section 5.2.1).

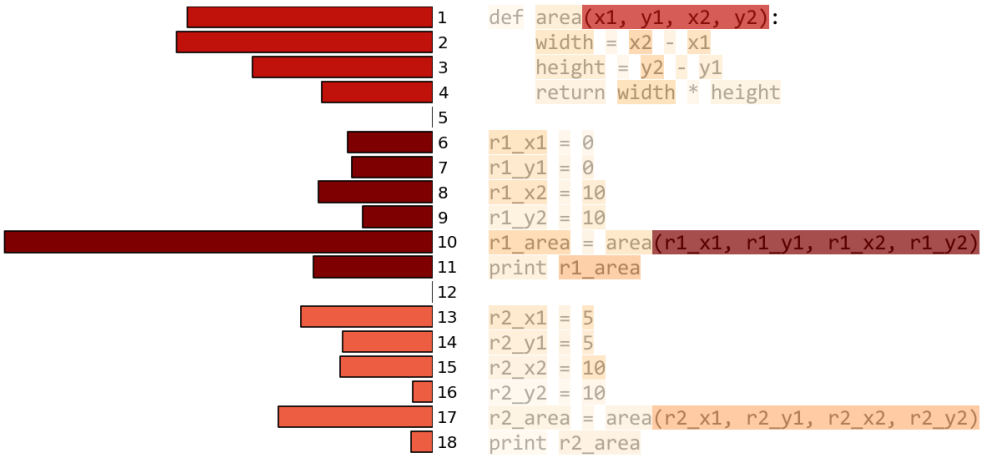


Figure 50: *Total fixation durations by line the `basic` version of `rectangle`.*

The `tuples` version was distinguishable by participants' use of their time on lines 2 and 3, especially around the `xy_2` operand. The extra time was likely used to verify the calculations of `width` and `height`, which could easily have been wrong if the indices provided to `xy_1` and `xy_2` were reversed. The corresponding lines in the `basic` and `class` versions were not fixated as much, suggesting that the calculation verification was easier with flat variable names (e.g., `x1`, `x2`).

The transition matrix for `tuples` exemplifies the "block" structure observed in the matrices of all three versions (Figure 53). Clusters of high transition probabilities closely follow the `area` function body and two area computations. Two additional transition probabilities stand out here: (1) from line 9 to 11, and (2) from the output box to line 11. These seem likely to correspond to different evaluation strategies – either reading all the way through the program or responding to the first area computation before continuing. Inspecting individual trial timelines, we can indeed find examples of both! Figure 54 shows two examples, with the highlighted regions corresponding to the participants' response periods. On top, we see a participant read the entire program from top to bottom before responding. On the bottom, there are distinct reading/response phases for both area computations. Note also that the `area` function (lines 1-4) is not fixated during the second computation; another example of short-term learning.

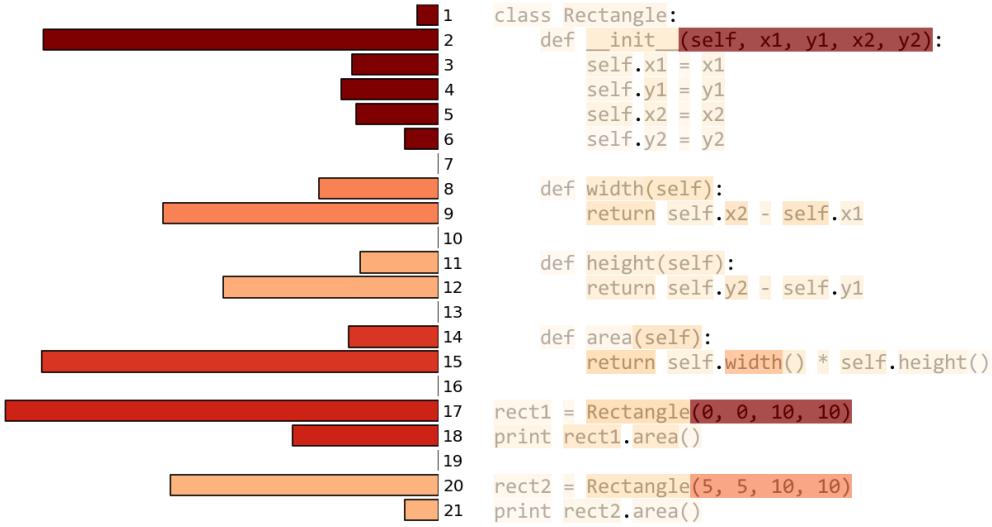


Figure 51: *Total fixation durations by line the* `class` *version of* `rectangle`*.*

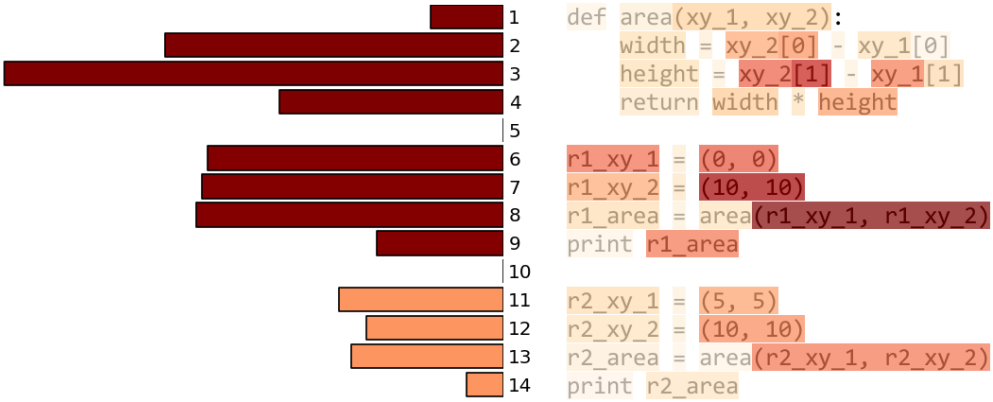


Figure 52: *Total fixation durations by line the* `tuples` *version of* `rectangle`*.*

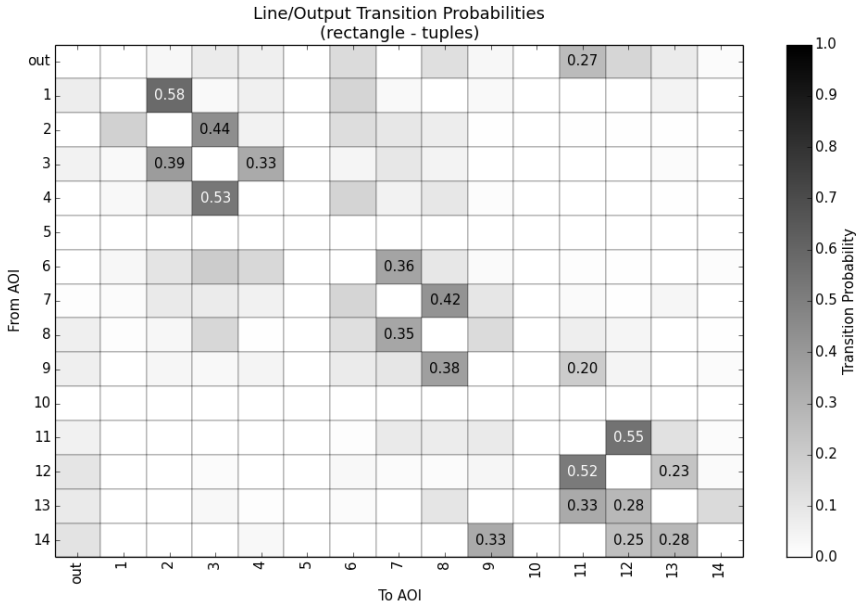


Figure 53: *Transition matrix for all* `tuples` *trials. Probabilities below 0.1 are not annotated.*

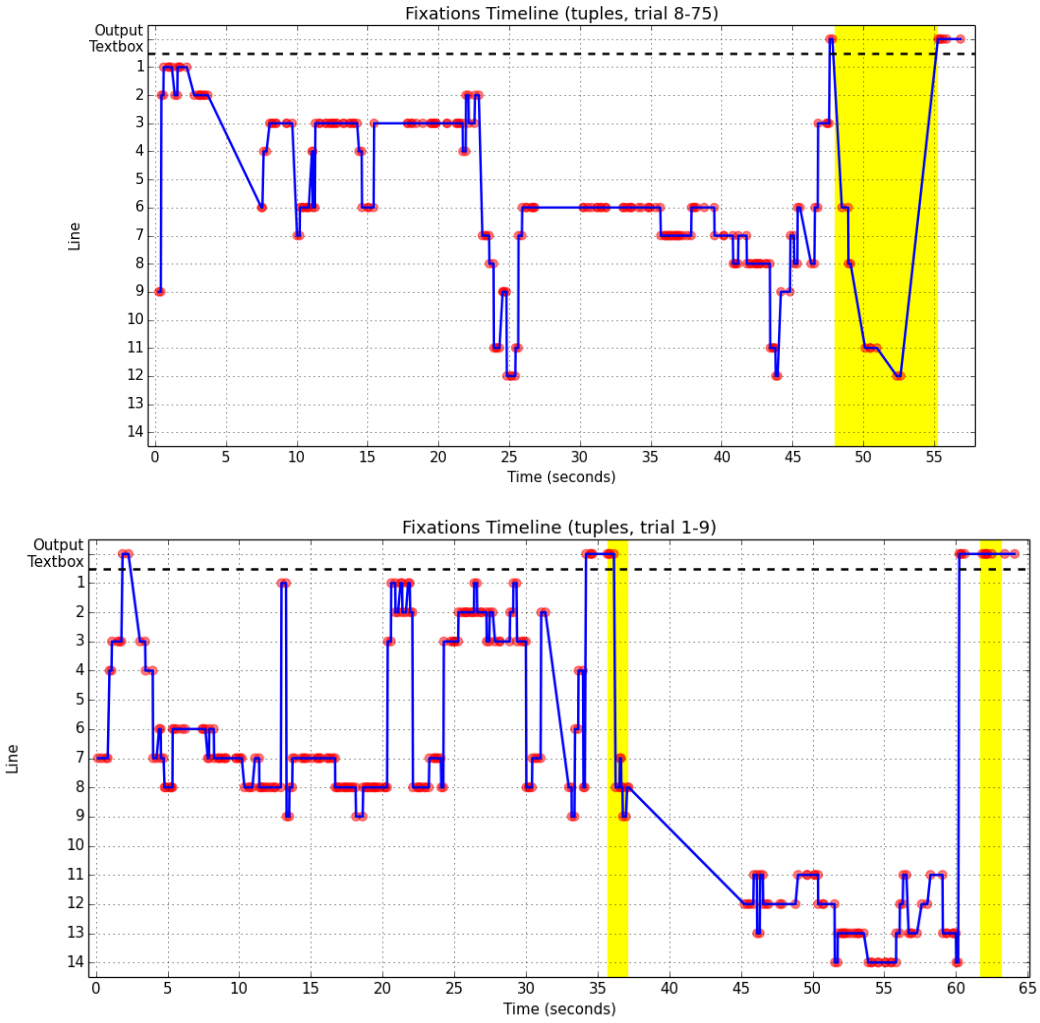


Figure 54: *Fixation timelines for two* `tuples` *trials. The yellow regions correspond to response periods.*

5.2.9 scope

The `scope` programs applied two functions to a variable named `added`: one function named `add_1`, and the other named `twice`. Both of these functions produced no visible effects – they did not actually modify their arguments or return a value. In the `samename` version, we reused the variable name `added` for each function's parameter name. For the `diffname` version, however, we used a different parameter name (`num`) for both functions. Because the `add_1` and `twice` functions had no effect, the main `added` variable retained its initial value of 4 throughout the program (instead of being 22).

Participants made errors on both versions of the `scope` programs at about the same rate, and we did not observe non-eye-tracking performance differences between versions (trial duration, response proportion, etc.). In the `diffname` version, however, significantly more fixation time was spent on line 2 relative to the rest of the program (see the bottom of Figure 55). In this version, a variable named `added` existed in the main program body **and** inside the `add_1` and `twice` functions. The extra time spent on line 2 suggests that having the same variable name for global and local `added` variables had a measurable effect, though not one that was visible by simply looking at responses and other non-eye-tracking performance metrics.

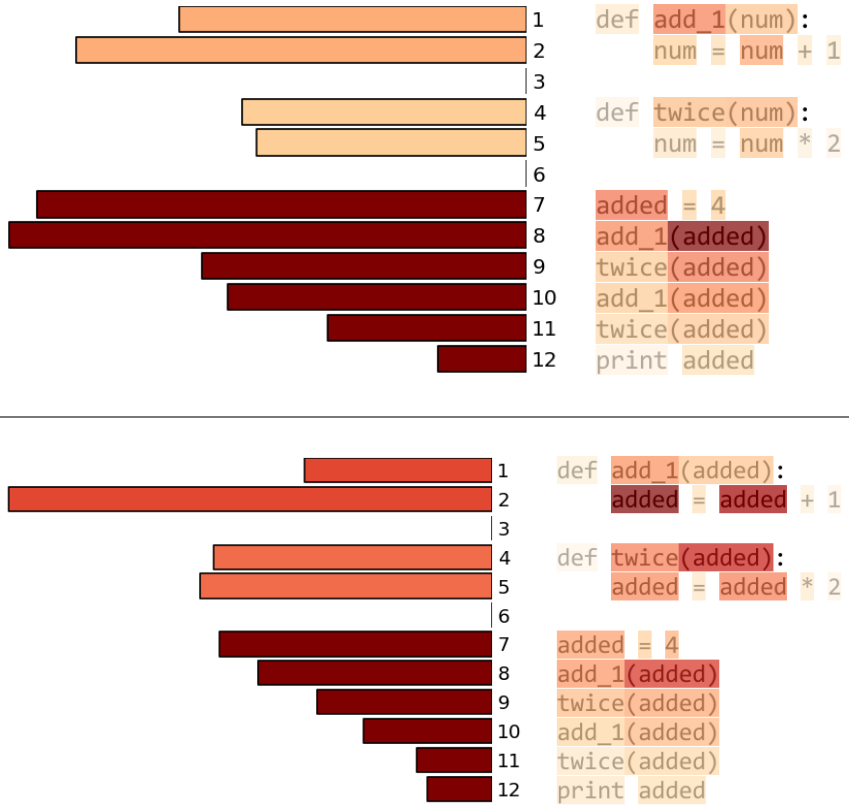


Figure 55: *Total fixation durations by line for both versions of the `scope` program (all participants). Top: `diffname`, bottom: `samename`.*

We did not observe any significant differences between versions in the transition matrices, and only minor differences existed in the transition matrices between correct and incorrect trials (Figure 56). Specifically, we

note an increased probability of returning from the output box to line 8, as well as transitioning from line 2 to line 1. The former may simply be an additional check by participants after responding, and the latter may be evidence of recognition that the modified variable (`added` or `num`) is locally rather than globally scoped. The true reasons for these differences, however, may be much more complicated. As we mentioned previously, some participants asked the experimenter if functions in the Python language were "call by value" or "call by reference" when evaluating this program. We did not expect such a simple-looking program to induce deep language-related questions, and make our interpretation of the data so difficult!

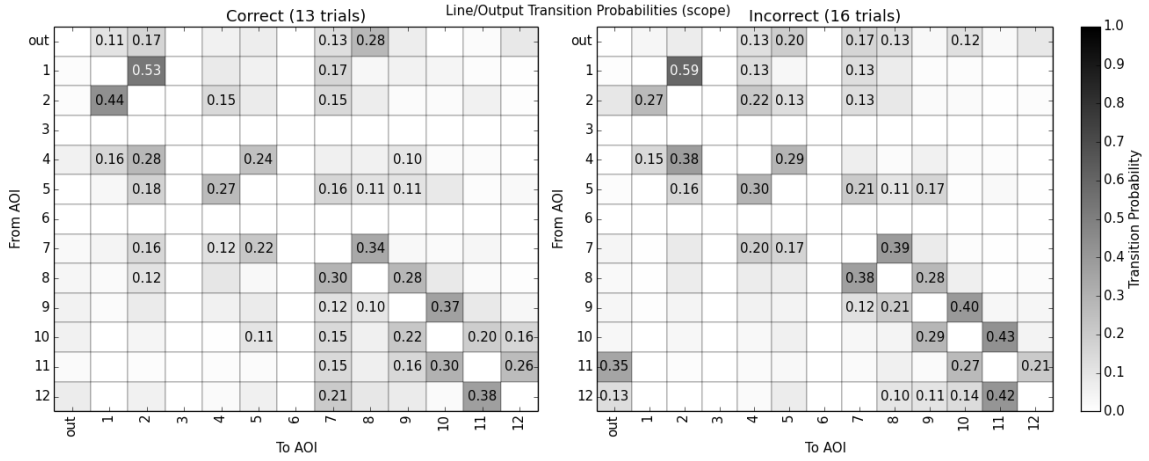


Figure 56: *Transition matrices for correct and incorrect trials in the* `scope` *programs. Probabilities below 0.1 are not annotated.*

5.2.10 whitespace

The `whitespace` programs print the results of three simple linear calculations. In the `zigzag` version, the code is laid out with one space between every mathematical operation, so that the line endings have a typical "zig-zag" appearance. The `linedup` version, in contrast, aligns each block of code by its mathematical operators, nicely lining up all identifiers. We expected there to be a speed difference between the two versions, with participants being faster in the `linedup` version. When designing the experiment, most of our pilot participants agreed that this version facilitated reading, but performance differences were not observed in practice.

We did not find differences in eye movements between versions either. Time spent on each line was approximately the same (Figure 57), and the average lengths of saccades as well as the spatial density of fixations on code were statistically indistinguishable. The transition matrices were also quite similar (Figure 58), leading us to conclude that lining up the text in this small program did not have an impact on eye movements. Because both styles are commonly found in code, programmers may have no trouble switching back and forth. Comparing one or both of these styles to a third, *uncommon* style may reveal differences (e.g., breaking the line in unusual places).

As with our programs, we observed correlations between fixation duration and specific times in a trial when we expected participants to be performing mental calculations. Figure 59 shows the fixation timeline

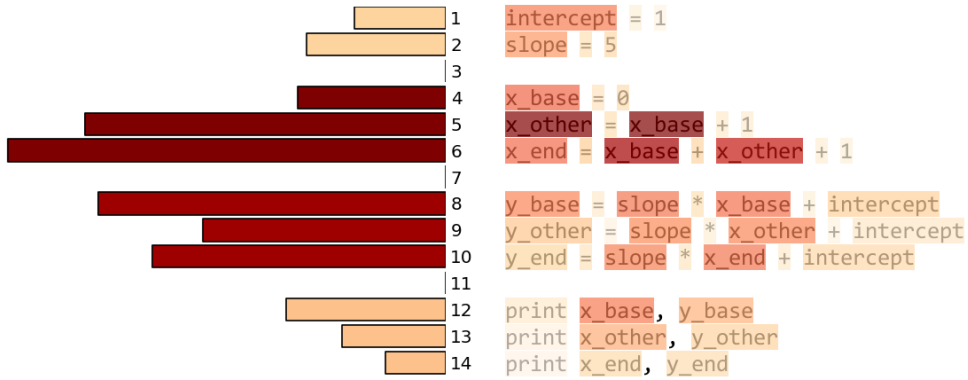


Figure 57: *Total fixation durations by line for both versions of the* `whitespace` *program (all participants). Top:* `linedup`*, bottom:* `zigzag`*.*

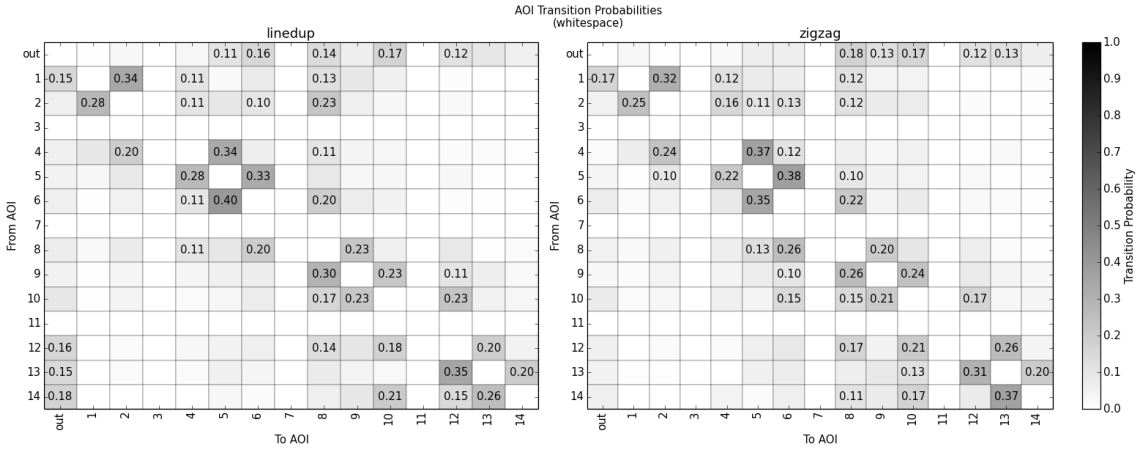


Figure 58: *Transition matrices for both versions of* `whitespace`*. Probabilities below 0.1 are not annotated.*

for a single `zigzag` trial (top) and the participant's average fixation duration over a rolling one second window every half second (bottom). The six highlighted regions correspond to the typing of the six parts of the response: 0, 1, 1, 6, 2, 11 (including spaces and new lines). Note that most of these regions overlap with local spikes in fixation duration, and are also places where we might expect the participant to be calculating. The `y` component calculations are relatively more difficult, and do indeed correspond to clear spikes. The final `x` component response (for `x_end`, however, also corresponds to a spike despite being a simple addition. We hypothesize that the increase in fixation duration may be due to mental *retrieval* of variable values (`x_base`, `x_other`) rather than calculation. A quantitative cognitive model of program comprehension could predict the underlying source of the fixation duration spikes even though they may produce the same observable phenomena.

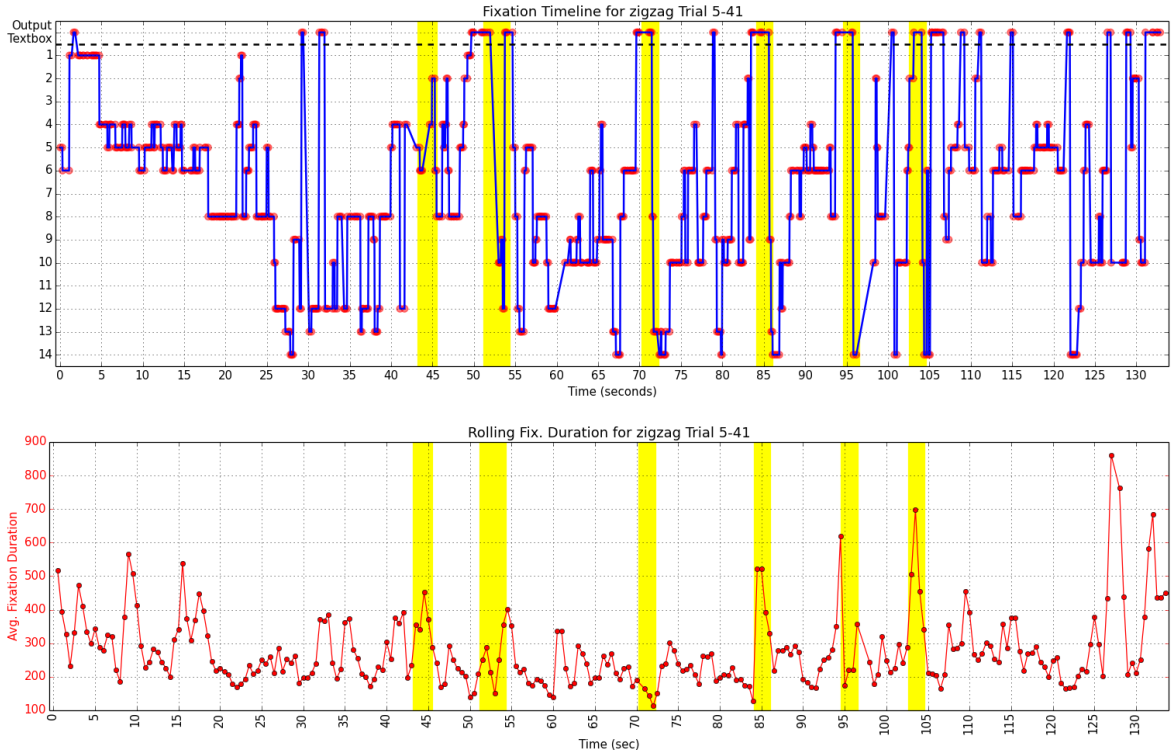


Figure 59

6 Discussion

Across 29 participants and 288 trials, we found programmers' eye movements to be strongly associated with the task at hand (output prediction), and highly informative about the relevant parts of each program. We saw evidence for short-term learning, both with repeated function calls and repeated code patterns. Surprisingly, we did not find large differences between the eye movements of more and experienced programmers. While our larger study with Mechanical Turk did find instances where performance differed for more experienced programmers, this did not show up in our eye-tracking analysis. Despite this, we did observe one case where eye movements and task performance were strongly linked: the `counting` programs. Below, we provide a high-level discussion of our results as they relate to expected task-oriented behavior by participants and our three main research questions.

6.1 Task-Oriented Behavior

Participants' eye movement data revealed strong task-specific signatures for some programs. While the data from all programs were indicative of task-oriented behavior, the distributions of fixation duration in `between`, `initvar`, and `rectangle` were especially insightful. In the `between` programs, participants spent less time looking at the *second instance* of the `between` calculation in both versions. Similar behavior was observed in `scope` and `whitespace` for repeated function calls and calculations, indicating short-term learning and algorithm recognition by participants.

In the `bothbad` version of `initvar`, a decrease in fixation duration on lines associated with the first calculation suggested that participants were short-circuiting the product after noticing the first term was zero. The timeline for one `bothbad` trial in Figure 40 and the corresponding keystrokes in Table 5 provided corroborating evidence, specifically an immediate response of "0" followed by an explicit calculation. On the flip side in `bothbad`, one trial's fixation timeline, average rolling fixation duration, and sequence of keystrokes signaled the participant's mental calculation (Figures 15 and 17). We found a number of cases where spikes in the average rolling fixation duration correlated with times when the participant was expected to be calculating (i.e., between looking at the calculation line and responding).

Fixations in all three versions of `rectangle` were heavily focused on function arguments, either to the `Rectangle` constructor or the `area` function. Because all participants responded correctly to this program, we considered the possibility that little to no verification of the program was done – i.e., the `area` calculation was simply assumed to be correct. The eye movement data tells a different story. **Both** the arguments passed to `area` or `Rectangle` and the arguments of the function/class definition were focused on. In the `tuples` version, focus shifted to the extraction of coordinate components for the rectangle's `width` and `height`, again indicating that participants were intent on verifying the program's behavior. The transition matrices from `rectangle` also hinted at multiple strategies that participants were using to evaluate the program. Further investigation revealed trials that did precisely this (Figure 54), demonstrating the power of high-level, aggregate eye movement statistics.

6.2 Research Questions

Next, we review our three research questions in light of the detailed results in the previous section.

R1. How does the eye movement data from our experiment compare to other eye-tracking program comprehension experiments?

Based on previous program comprehension experiments with eye-tracking, we had specific expectations for our low-level eye movement metrics. Average fixation durations in previous experiments were higher than ours (300-400 ms versus 273 ms), and were also correlated with the participant's programming experience. We did not find programming or Python experience to be even moderately correlated with any of our eye-tracking metrics (both low and high level). We attribute these differences to the uniqueness of our task and the relative simplicity of our programs. Other experiments have had participants debug more complex code, such as binary search algorithms and prime number generators. In contrast, one of our most "complex" programs, `between`, featured list filtering and intersection. Such simple programs may not invoke the same behaviors observed in other experiments, especially in more experienced programmers.

Although our fixation metrics differed from other experiments, the distributions of fixation duration *across areas of interest* produced results more in line with expectations. Keywords, for example, received the least amount of focus, while more complex expressions with conditions and operators received the most. A perfect comparison between experiments is not possible, however, due to differences in how some code elements are categorized. We consider Python's `True` and `False` to be literal values like numbers and strings, but they are categorized as keywords in at least one other Java-based study [8]. Similar problems exist when comparing code complexity metrics like Halstead volume [17] between programming languages. Precisely defining which code elements are "operators", "operands", or neither can be surprisingly difficult – e.g., is the semi-colon in Java an end-of-statement operator?

R2. Can aggregate eye movement metrics and summary statistics be predicted from textual/syntactic features of code?

Across all programs and participants, we found that line-based reading behavior could be moderately predicted from a few simple textual features. The total amount of fixation duration on a line, for example, was decently predicted ($r^2 = 0.54$) from just the length of the line and its proportion of whitespace characters. Similarly, the time of the first fixation on a line (divided by the total trial duration) could be predicted fairly well ($r^2 = 0.57$) from the line number alone. Surprisingly, the category of a line – e.g., function call, `if` statement, `for` loop – did not significantly improved predictions. At the level of individual code elements, however, the category was useful in predicting total fixation duration. A simple linear model with the categories `keyword`, `list`, `identifier`, `operator`, `string`, `integer`, and `tuple` achieved an r^2 value of 0.383. When the line number of the code element was included, model performance increased to $r^2 = 0.576$. Thus, it appears that the category of a code element, and not the entire line, is a better predictor of fixation duration, though line number still plays a significant role.

Several patterns emerged from the AOI transition matrices for individual programs. High probability transitions tended to be clustered around whitespace-separated blocks of code, with transitions up and down the block. The matrix in Figure 53 for `rectangle tuples` is an especially clean example of this pattern. We expected the highest probability transitions from code to the output box to come from `print` statements. While this was simply true for some programs, such as `funcall` (Figure 35), the results were more complicated for most others. The transition matrices for `initvar` shown in Figure 39, for example, require additional explanation. In the `good` version (left), transitions to the output box tend to come from the

final `print` statement (0.33 probability). The sum of probabilities from lines 2-4 to the output box, however, accounts for just as much. Transitions *from* the output box often went to lists, such as those on lines 15 and 19 in `between` `functions` (Figure 27). This makes sense, given that participants were supposed to filter the lists and type their responses. In fact, we found a moderate correlation ($r = 0.30$, $p < 0.001$) between the transition probability from the output box for a given line and the total fixation duration on that line. If we interpret fixation duration as a measure of line importance, this says that participants tend to return to important lines after providing partial responses (which almost always co-occurred with fixating the output box).

R3. Do differences between versions of the same program, or demographics/performance of the participant, influence eye movements?

We did not observe strong differences in low-level eye movement metrics or high-level statistics between versions for most of our programs. Specifically, the two or three versions of `between`, `funcall`, `order`, `partition`, `scope`, and `whitespace` were very similar in terms of fixation duration distribution and AOI transition probabilities. There were not significant *performance* differences between participants interpreting different versions either in this study (though there were in our larger study).

The different versions of `counting`, `initvar`, `overload`, and `rectangle`, however, were distinguishable by eye movement metrics and statistics. For `counting`, we observed a distinct difference between the AOI transition matrices of correct and incorrect trials in `twospaces` version. Participants who gave correct responses tended to read all lines of the program before responding, while the others waited to read the final line. As mentioned above, the first calculation in the `bothbad` version of `initvar` received less focus, presumably because participants recognized a shortcut in the calculation not present in the other two versions. In the `plusmixed` and `multmixed` versions of `overload`, the final "5" + "3" operation received more time than in the `strings` version where all previous + operations were also string concatenations. Finally, the `tuples` version of `rectangle` demonstrated a unique allocation of fixation duration on the calculation of the rectangle's width and height – a point where correct tuple indexing was critical.

Surprisingly, we did not find a strong or even moderate correlation between a participant's programming experience, Python experience, or response correctness and **any** of our low-level eye movement metrics. Programming experience was a statistically significant predictor in a simple linear model predicting average fixation duration. However, the effect size (r^2) was extremely small (< 0.1), so we do not have much confidence in its generalizability to other groups of programmers. As we discussed earlier for research question R1, our unique task and simple programs may account for these results. Other performance metrics, such as trial duration, were trivially correlated with number of fixations in a trial and total scanpath length – longer trials necessarily mean more fixations and saccades. Other eye movement metrics, like average fixation duration, fixation rate, and average saccade length were not correlated with any performance metric or demographic value.

6.3 Scanpath Comparison

We had hoped to use the Levenshtein (string-edit) distance as a means of comparing participant scanpaths on the same program. This proved more difficult than expected, as individual scanpaths were quite different, even at multiple levels of granularity. While scanpaths at the code element level (e.g., keywords, identifiers)

will be obviously different and quite noisy, we were surprised to find similar obstacles at the line and whitespace-separated block level. On average, we found differences of over 50% between line and block scanpaths (over 60% for line). These results suggest that alternative techniques may be necessary for meaningful comparisons of programmer scanpaths, even for very simple programs. For example, Cristino et. al have developed a technique based on the Needleman-Wunsch algorithm used to compare DNA sequences in bioinformatics [12]. Hayes et. al have also created a method for converting scanpaths to a type of transition matrix that retains temporal information (called the scanpath successor representation) [19]. These new techniques may provide additional insight, but they are beyond the scope of this work.

7 Conclusion

Eye movements are a rich data source, and our analysis has shown just how much information can be extracted from a mere 29 participants. Over the course of 288 trials, our participants predicted the output of 25 total programs (10 programs each). Low-level eye movement metrics, such as fixation duration and saccade length, did not correlate with task performance or programming experience. Fixation duration did serve as an excellent proxy for code line importance, with the task-relevant lines of a program often receiving the most aggregate focus. In line with previous experiments, we found that code elements like keywords and identifiers received the least amount of focus, while more complex statements involving conditional/boolean expressions received more.

High-level eye movement metrics and plots provided the most insight into our programmers' cognitive processes. Area of interest (AOI) transition matrices revealed how often participants transitioned between code lines and the output box. In several cases, these transition probabilities provided hints at the kinds of strategies being used to evaluate the programs. In the `counting` programs, for example, correct and incorrect responses to the `twospaces` version had distinct transition matrices, with the latter having a smaller probability of reading the entire program before starting to respond. The transition matrix for `rectangle tuples` had transitions corresponding to two different strategies (single response, multi-response), and we found trials clearly demonstrating each. Lastly, we combined high and low-level metrics using a rolling time window across individual trials. When looking at combined changes in mean fixation duration, recently fixated lines, and response keystrokes, we found that spikes in fixation duration often co-occurred with times we would expect the participant to perform a mental calculation.

Eye-tracking has recently gained popularity in the program comprehension literature, augmenting or replacing traditional methodologies like think-aloud. Metrics and plots derived from eye movement data can be used to gain invaluable insight into a programmer's cognitive processes. This data could also be used to develop a cognitive model of program comprehension, a step towards the semi-automated analysis of program language design. We plan to use the data collected in this experiment to create a functioning version of Cant et. al's Cognitive Complexity Metric [9]. By incorporating existing components of a *cognitive architecture* – a computational model of human cognitive – we will build on years of modeling research in human memory and perceptual/motor systems.

7.1 Future Work

Aside from developing a cognitive model, there are many other interesting avenues for future work. An obvious extension to our experimental methodology is to include additional programming languages. Java is used in a number of other studies, and so is a likely candidate. More complex programs may elicit differences between more and less experienced programmers, but may require introducing the skeleton of a development environment. A multi-file program, for example, would require additional GUI elements like tabs or a file explorer. Human-computer interface studies on integrated development environments (IDEs) exist (e.g., [22]), and it would be interesting to add eye-tracking to the mix.

We used rolling metrics to correlate spikes in fixation duration with moments where we expected participants to be performing some kind of mental calculation or memory retrieval. Many other eye movement metrics, such as spatial density and saccade length, could be analyzed in a similar manner. Additionally, different window sizes focus on different timescales, providing yet another analysis parameter to vary.

Finally, our comparison of scanpaths was stunted by the brittle nature of the Levenshtein distance. More robust, genomic-inspired methods, like ScanMatch [12] are in our sights for future work. A more radical approach would involve starting at our AOI mapping stage, where fixations are assigned to specific AOIs. Rather than picking a single AOI, the areas of overlap between a circle surrounding the fixation point and proximate AOIs could produce multiple possible scanpaths. Scanpath comparisons would then be between multi-dimensional spaces, rather than one dimensional series, of AOIs.

8 Acknowledgements

Grant R305A1100060 from the Institute of Education Sciences Department of Education and grant 0910218 from the National Science Foundation REESE supported this research.

A Appendix - Programs and Output

A.1 between

A.1.1 between - functions

```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

def common(list1, list2):
    winners = []
    for item1 in list1:
        if item1 in list2:
            winners.append(item1)
    return winners

x = [2, 8, 7, 9, -5, 0, 2]
x_btwn = between(x, 2, 10)
print x_btwn

y = [1, -3, 10, 0, 8, 9, 1]
y_btwn = between(y, -2, 9)
print y_btwn

xy_common = common(x, y)
print xy_common
```

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

A.1.2 between - inline

```
x = [2, 8, 7, 9, -5, 0, 2]
x_between = []
for x_i in x:
    if (2 < x_i) and (x_i < 10):
        x_between.append(x_i)
print x_between

y = [1, -3, 10, 0, 8, 9, 1]
y_between = []
for y_i in y:
    if (-2 < y_i) and (y_i < 9):
        y_between.append(y_i)
print y_between

xy_common = []
for x_i in x:
    if x_i in y:
        xy_common.append(x_i)
print xy_common
```

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

A.2 counting

A.2.1 counting - nospace

```
for i in [1, 2, 3, 4]:
    print "The count is", i
    print "Done counting"
```

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

A.2.2 counting - twospaces

```
for i in [1, 2, 3, 4]:
    print "The count is", i


    print "Done counting"
```

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

A.3 funcall

A.3.1 funcall - nospace

```
def f(x):
    return x + 4

print f(1)*f(0)*f(-1)
```

```
60
```

A.3.2 funcall - space

```
def f(x):
    return x + 4

print f(1) * f(0) * f(-1)
```

```
60
```

A.3.3 funcall - vars

```
def f(x):
    return x + 4

x = f(1)
y = f(0)
z = f(-1)
print x * y * z
```

```
60
```

A.4 initvar

A.4.1 initvar - bothbad

```
a = 0
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
0
11
```

A.4.2 initvar - good

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 0
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
24
10
```

A.4.3 initvar - onebad

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
24
11
```

A.5 order

A.5.1 order - inorder

```
def f(x):
    return x + 4

def g(x):
    return x * 2

def h(x):
    return f(x) + g(x)

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
5 2 7
```

A.5.2 order - shuffled

```
def h(x):
    return f(x) + g(x)

def f(x):
    return x + 4

def g(x):
    return x * 2

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
5 2 7
```

A.6 overload

A.6.1 overload - multmixed

```
a = 4
b = 3
print a * b

c = 7
d = 2
print c * d

e = "5"
f = "3"
print e + f
```

```
12
14
53
```

A.6.2 overload - plusmixed

```
a = 4
b = 3
print a + b

c = 7
d = 2
print c + d

e = "5"
f = "3"
print e + f
```

```
7
9
53
```

A.6.3 overload - strings

```
a = "hi"
b = "bye"
print a + b

c = "street"
d = "penny"
print c + d

e = "5"
f = "3"
print e + f
```

```
hibye
streetpenny
53
```

A.7 partition

A.7.1 partition - balanced

```
for i in [1, 2, 3, 4, 5]:
    if (i < 3):
        print i, "low"
    if (i > 3):
        print i, "high"
```

```
1 low
2 low
4 high
5 high
```

A.7.2 partition - unbalanced

```
for i in [1, 2, 3, 4]:
    if (i < 3):
        print i, "low"
    if (i > 3):
        print i, "high"
```

```
1 low
2 low
4 high
```

A.7.3 partition - unbalanced_pivot

```
pivot = 3
for i in [1, 2, 3, 4]:
    if (i < pivot):
        print i, "low"
    if (i > pivot):
        print i, "high"
```

```
1 low
2 low
4 high
```

A.8 rectangle

A.8.1 rectangle - basic

```
def area(x1, y1, x2, y2):
    width = x2 - x1
    height = y2 - y1
    return width * height

r1_x1 = 0
r1_y1 = 0
r1_x2 = 10
r1_y2 = 10
r1_area = area(r1_x1, r1_y1, r1_x2, r1_y2)
print r1_area

r2_x1 = 5
r2_y1 = 5
r2_x2 = 10
r2_y2 = 10
r2_area = area(r2_x1, r2_y1, r2_x2, r2_y2)
print r2_area
```

```
100
25
```

A.8.2 rectangle - class

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def width(self):
        return self.x2 - self.x1

    def height(self):
        return self.y2 - self.y1

    def area(self):
        return self.width() * self.height()

rect1 = Rectangle(0, 0, 10, 10)
print rect1.area()

rect2 = Rectangle(5, 5, 10, 10)
print rect2.area()
```

```
100
25
```

A.8.3 rectangle - tuples

```
def area(xy_1, xy_2):
    width = xy_2[0] - xy_1[0]
    height = xy_2[1] - xy_1[1]
    return width * height

r1_xy_1 = (0, 0)
r1_xy_2 = (10, 10)
r1_area = area(r1_xy_1, r1_xy_2)
print r1_area

r2_xy_1 = (5, 5)
r2_xy_2 = (10, 10)
r2_area = area(r2_xy_1, r2_xy_2)
print r2_area
```

```
100
25
```

A.9 scope

A.9.1 scope - diffname

```
def add_1(num):
    num = num + 1

def twice(num):
    num = num * 2

added = 4
add_1(added)
twice(added)
add_1(added)
twice(added)
print added
```

```
4
```

A.9.2 scope - samename

```
def add_1(added):
    added = added + 1

def twice(added):
    added = added * 2

added = 4
add_1(added)
twice(added)
add_1(added)
twice(added)
print added
```

```
4
```

A.10 whitespace

A.10.1 whitespace - linedup

```
intercept = 1
slope     = 5

x_base  = 0
x_other = x_base + 1
x_end   = x_base + x_other + 1

y_base  = slope * x_base  + intercept
y_other = slope * x_other + intercept
y_end   = slope * x_end   + intercept

print x_base,  y_base
print x_other, y_other
print x_end,   y_end
```

```
0 1
1 6
2 11
```

A.10.2 whitespace - zigzag

```
intercept = 1
slope = 5

x_base = 0
x_other = x_base + 1
x_end = x_base + x_other + 1

y_base = slope * x_base + intercept
y_other = slope * x_other + intercept
y_end = slope * x_end + intercept

print x_base, y_base
print x_other, y_other
print x_end, y_end
```

```
0 1
1 6
2 11
```

References

[1] Roman Bednarik. *Methods to analyze visual attention strategies: Applications in the studies of programming*. University of Joensuu, 2007.

[2] Roman Bednarik, N. Myller, E. Sutinen, and M. Tukiainen. Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, pages 66–82, 2006.

[3] Roman Bednarik and Markku Tukiainen. Effects of display blurring on the behavior of novices and experts during program debugging. In *CHI'05 Extended abstracts on human factors in computing systems*, pages 1204–1207. ACM, 2005.

[4] Roman Bednarik and Markku Tukiainen. Temporal eye-tracking data: evolution of debugging strategies with multiple representations. In *Proceedings of the 2008 symposium on Eye tracking research & applications*, pages 99–102. ACM, 2008.

[5] Georg Brandl and Pygments contributers. Pygments. `http://www.pygments.org`, Sep 2014.

[6] Raymond Buse and Wes Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558, 2010.

[7] Teresa Busjahn. Personal communication, 2014.

[8] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9. ACM, 2011.

[9] Simon Cant, David Jeffery, and Brian Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, 1995.

[10] Roger HS Carpenter. *Movements of the eyes (2nd rev)*. Pion Limited, 1988.

[11] Laura Cowen, Linden Js Ball, and Judy Delin. An eye movement analysis of web page usability. In *People and Computers XVI-Memorable Yet Invisible*, pages 317–335. Springer, 2002.

[12] Filipe Cristino, Sebastiaan Mathôt, Jan Theeuwes, and Iain D Gilchrist. Scanmatch: A novel method for comparing fixation sequences. *Behavior Research Methods*, 42(3):692–700, 2010.

[13] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[14] Françoise Détienne. What model (s) for program understanding? In *Proceedings of the Conference on Using Complex Information Systems*, Poitiers, France, 1996.

[15] Françoise Détienne and Frank Bott. *Software design–cognitive aspects*. Springer Verlag, 2002.

[16] Christopher Douce. The stores model of code cognition. In *Psychology of Programming Interest Group*, 2008.

[17] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[18] Michael Hansen, Robert L. Goldstone, and Andrew Lumsdaine. What Makes Code Hard to Understand? *ArXiv e-prints*, April 2013.

[19] Taylor R. Hayes, Alexander A. Petrov, and Per B. Sederberg. A novel method for analyzing sequential eye movements reveals strategic influence on raven's advanced progressive matrices. *Journal of vision*, 11(10), 2011.

[20] Ignace Hooge and Guido Camps. Scan path entropy and arrow plots: capturing scanning behavior of multiple observers. *Frontiers in psychology*, 4, 2013.

[21] Eric Jones, Travis Oliphant, Pearu Peterson, and Others. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–. [Online; accessed 2014-10-10].

[22] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

[23] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

[24] Janni Nielsen, Torkil Clemmensen, and Carsten Yssing. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110. ACM, 2002.

[25] Marcus Nyström, Richard Andersson, Kenneth Holmqvist, and Joost van der Wijer. The influence of calibration method and eye physiology on eyetracking data quality. *Behavior research methods*, 45(1):262–288, 2013.

[26] A Olsen. The tobii i-vt fixation filter: Algorithm description. *Tobii I-VT Fixation Filter–whitepaper [White paper]*, 2012.

[27] Alex Poole and Linden J. Ball. Eye tracking in HCI and usability research. *Encyclopedia of Human-Computer Interaction, C. Ghaoui (ed.)*, 2006.

[28] Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

[29] Dario D. Salvucci. *Mapping eye movements to cognitive processes*. PhD thesis, Carnegie Mellon University, 1999.

[30] David Sankoff and Joseph B. Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. *Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B.*, 1, 1983.

[31] Ben Schneiderman and Richard Mayer. Towards a cognitive model of programmer behavior. Technical report, Indiana University, aug 1975.

[32] J.S. Seabold and J. Perktold. Statsmodels: Econom tric and statistical modeling with pythong. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, 2010.

[33] Kshitij Sharma, Patrick Jermann, Marc-Antoine Nüssli, and Pierre Dillenbourg. Gaze evidence for different activities in program understanding. In *24th Psychology of Programming Workshop*, 2012.

[34] Sidney Siegel and N. John Castellan. *Nonparametric statistics for the behavioral sciences*. McGraw–Hill, Inc., second edition, 1988.

[35] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[36] Tobii Eye Tracking Research. Tobii Studio 3.2 User Manual. `http://www.tobii.com/Global/Analysis/Downloads/User_Manuals_and_Guides/Tobii_UserManual_TobiiStudio3.2_301112_ENG_WEB.pdf`, Jun 2014.

[37] W.J. Tracz. Computer programming and the human thought process. *Software: Practice and Experience*, 9(2):127–137, 1979.

[38] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140. ACM, 2006.

[39] Adrian Veßkühler. OGAMA. `http://www.ogama.net`, Sep 2014.

[40] Hans W. Wendt. Dealing with a common problem in social science: A simplified rank-biserial coefficient of correlation based on the u statistic. *European Journal of Social Psychology*, 2(4):463–465, 1972.