Eye-movement indices of reading while debugging Python source code

[1]Jack Dempsey, [2]Anna Tsiola, [1,3]Nigel Bosch, [1,4]Kiel Christianson, and [5]Mallory Stites

[1]Department of Educational Psychology, University of Illinois Urbana–Champaign, Champaign, IL, USA

[2]Department of Linguistics, University of Illinois Urbana–Champaign, Champaign, IL, USA

[3]School of Information Sciences, University of Illinois Urbana–Champaign, Champaign, IL, USA

[4]Beckman Institute for Advanced Science & Technology, University of Illinois Urbana–Champaign, Urbana, IL, USA

[5]Sandia National Laboratories, Albuquerque, NM, USA

November 2024

Corresponding author:

Jack Dempsey

Department of Educational Psychology
Education Building, Rm. 226A
University of Illinois
1310 S. 6th St.
Champaign, IL 61820

E-mail: <jkdemps2@illinois.edu>
Dept. Tel: 217.333.2245

ABSTRACT

Unlike text reading, the eye-movement behaviors associated with reading Python, a computer programming language, are largely understudied through a psycholinguistic lens. A general understanding of the eye movements involved in reading while troubleshooting Python, and how these behaviors compare to proofreading text, is critical for developing educational interventions and interactive tools for helping programmers debug their code. These data may also highlight to what extent humans use their underlying text reading ability when reading source code. The current work provides a profile of global reading behaviors associated with reading Python source code for debugging purposes. To this end, we recorded experienced programmers' eye movements while they determined whether 21 different Python functions would produce the desired output, an incorrect output, or an error message. Some reading behaviors seem to mirror those found in text reading (e.g., effects of stimulus complexity), while others may be specific to reading code. Results suggest that semantic errors that produce undesired outputs in programming source code may influence early stages of processing, likely due to the largely top-down strategy employed by experienced programmers when reading source code. The findings are framed to invigorate discussion and further exploration into psycholinguistic analysis of human source code reading.

INTRODUCTION

When we think about reading in natural contexts, what usually comes to mind is a series of sentences and paragraphs meaningfully connected in a discourse. For example, we might think of reading a novel, reading instructions for assembling furniture, reading comments on an internet video, or reading a menu at a restaurant, among other activities. Although the cognitive approaches to reading these different types of print may differ, each task recruits processes endowed by the naturally developed human language faculty. Beyond this, there are other capacities that are often required to successfully read natural texts and extract contextually appropriate meaning. Computer programs are an example of texts that require both reading abilities and specialized knowledge to be able to interpret, and they are used by millions of people around the world.

As educational programs continue developing curricula designed to increase computer and programming literacy, it is important to consider exactly what this kind of literacy would mean (Vee, 2017). When programmers write functions using source code—for example, Java or Python—they are essentially writing instructions for a computer to complete some task given some information. According to the Theory of Dual Channel Constraints proposed by Casalnuovo and colleagues (2020), humans take advantage of two distinct sources of information when reading code: the algorithm channel and the natural language channel. The algorithm channel requires an understanding of the programming language being used (e.g., Python) and a knowledgebase that computers have access to; however, the natural language channel is supported by human reading ability, which therefore is not available for computers in processing source code. This distinction highlights a major difference between programming languages and human languages: syntactic constraints are typically less flexible in programming languages,

whereas humans use heuristics and real-time strategies to interpret erroneous or complex input (e.g., Christianson, 2016).

Following its relative lack of structural flexibility, studies in the past decade have also illustrated that source code is often more predictable than natural language text (Casalnuovo et al., 2019, 2020abc; cf. Luke & Christianson, 2016), and subsequent investigations have shown that programmers prefer predictable code (Casalnuovo et al., 2020c) and more efficiently comprehend it (Casalnuovo et al., 2020b), as evidenced by offline judgments. These patterns are similar to how predictability correlates with early reading behaviors for natural texts. Like reading natural texts, reading source code can be framed as a cognitive exercise, likely as a top-down hypothesis-driven process that differs in some key aspects from reading natural texts (Brooks, 1983; Schneiderman & Mayer, 1979). This top-down process is plausibly developed over years of experience in programming (Shaft & Vessey, 1995), meaning beginner programmers likely use bottom-up processes, similar to those recruited for language processing during reading, more often. It may seem somewhat trivial, then, that reading source code uses at least some subset of the same cognitive processes used in reading; however, it is an important area of research to better understand exactly where these two cognitive processes converge and diverge (Fedorenko et al., 2019).

To answer this question, it is first important to consider how cognitive processes are captured in the eye-movements of readers while they read natural texts. For over half a century, eye-tracking studies have shown that how long a reader fixates on a word, whether someone even fixates on a word, and whether someone goes back to previous material after reading a given word, among many other measures, are all positively correlated with cognitive processing difficulty (for a review of eye-movements in reading research, see Clifton et al., 2016). This line

of work depends on the so-called eye-mind link (e.g., Reichle & Reingold, 2013), which posits that the cognitive processes of lexical access and retrieval are what trigger most eye-movements during reading. Thus, characteristics of eye-movements reveal aspects of cognitive processes and can be used to test many psycholinguistic questions of interest, like how different syntactic structures are processed (e.g., Frazier & Rayner, 1982), how structural ambiguity is handled in real-time (e.g., Traxler et al., 1998), how lexical characteristics like word length, frequency, and predictability influence retrieval speed and quality of lexical representations during reading (e.g., Clifton et al., 2016; Rayner, 1998), or even what the overall complexity of a text is (Rayner et al., 2006). In sum, examining eye-movement patterns when reading certain forms of texts and in certain tasks can help us understand the cognitive processes recruited for those tasks.

For this study in particular, processing difficulty stems from either syntactic or semantic sources, likened to the syntactic and semantic errors encountered in natural reading. Syntactic and semantic error sources typically produce differential neural responses (Kutas & Hillyard, 1984; Osterhout & Holcomb, 1992), and semantic error sources often lead to later stages of processing that involve integration and text wrap-up (e.g., Payne & Stine-Morrow, 2014). Syntactic processing errors, on the other hand, have been shown to occur at earlier stages like gaze duration and go-past times (the time it takes to read a word and reread previous material before moving past said word, e.g., Frazier & Rayner, 1982), suggesting they are resolved or at least detected at earlier processing stages in natural text reading.

Recent work has sought to address the differences and similarities between reading of natural texts and source code reading, although it is not entirely new. For example, Pennington (1987) argued that programs are first procedurally understood before individual meaningful chunks, which means that reading source code may be less incrementally structured compared to

reading natural text. In terms of actual eye-movement behaviors, recent evidence suggests that initial code segments are read longer than segments that appear later (Jbara & Feitelson, 2017), there are more vertical eye movements when reading source code (Busjahn et al., 2015), and experience increases both the rate of vertical eye movements (Turner et al., 2014) and time spent looking at task-relevant code (Crosby et al., 2002; Peitek et al., 2020) and iterative loops (Herman et al., 2021). In a similar line of work looking at the reading of mathematical proofs, Inglis and Alcock (2012) found that experienced mathematicians spent more time moving their eyes between different lines of proofs than undergraduate students, suggesting experience drives these readers, like in the reading of source code, to focus more on integration between lines rather than surface features of the lines themselves. Furthermore, these changes in reading behaviors as programmers gain more experience are likely driven by some form of error-based learning; for example, vertical eye movements when reading source code have been shown to predict faster bug detection (Sharif et al., 2012), and this is also a characteristic of more experienced programmers' reading of source code.

Despite these recent advancements, there remain many unanswered questions regarding how experienced programmers read source code, although some important work has been conducted in the field of eye-movements in programming (EMIP). Perhaps most notably, Bednarik and colleagues (2020) established an eye-movement in programming dataset that is publicly available. Although also looking at an object-oriented language, most of the measures collected differ from measures typically studied in the field of reading psychology and instead focus on more global aspects of attention like saccade distance, average fixation duration, and saccade amplitude, among others. Moreover, the dataset is comprised of only two items, which limits its generalizability. A complementary dataset of more psycholinguistic eye-movement

measures while reading source code could help move this research effort along. In particular, there has yet to be 1) a descriptive psycholinguistic analysis of typical eye-movements when reading source code for debugging, or 2) an analysis of how eye-movements around erroneous code reflect debugging comprehension processes. The current study seeks to address this gap in knowledge, focusing specifically on experienced programmers' reading of Python source code while debugging[1].

Python is an imperative programming language, meaning that programmers write sequences of instructions for the computer to achieve the desired outcome. The code that Python programmers write is syntactically inflexible; for computers to successfully run the code in the expected manner, the exact characters must be entered. A line of code is lexically constrained in that the name of the function (e.g., *len*()) does not vary, while the variable it acts on can be given a virtually infinite number of different labels (e.g., "qwerty" instead of "word"). Python contains many functions that are disproportionately based on English words as well as programmer-defined functions named with content words chosen by the programmer. Although these content words could be almost anything, Python and other source code tends to be written by humans with other humans in mind (Knuth, 1984), meaning that content words in source code are often tailored to be easily understood both by the author of the code and other potential readers. An example Python function is shown in Figure 1 below.

[1] For the current study, we refer to "debugging" to mean reading through source code to identify potential errors. This is different from the typical debugging process where programmers may also test run code as a means of better identifying errors. Therefore, the current study is concerned with the general reading patterns associated with this specific earlier stage in the typical debugging process.

Figure 1. Example of a programmer-defined Python function that takes a string of characters as input and tests whether that string is a palindrome.

```
def is_palindrome(word):
    #This function determines whether a word is the same spelled backwards (i.e. a palindrome).
    #Example: Input: 'racecar'
    #         Output: True
    order = []
    rev_order = []
    for i in range(len(word)):
        order.append(word[i])
        rev_order.append(word[-(i+1)])
    if order == rev_order:
        return True
    else:
        return False
```

Debugging source code is a very common practice in computer programming, whether the code was originally written by the reader or by someone else, and can essentially be viewed as a type of proofreading in some capacity - a task which has been shown to elicit different eye-movement and reading behaviors compared to natural reading for comprehension. For example, Kaakinen and Hyöna (2010) showed that reading for proofreading resulted in more leftward landing positions, shorter saccade lengths, longer fixation durations, and a higher probability of rereading, broadly defined as fixating input that has already been fixated, compared to reading for comprehension. Schotter and colleagues (2014) further developed these findings by showing that different types of proofreading elicited different reading patterns. For example, when readers are proofreading for spelling, they show heightened sensitivity to frequency effects that have commonly been reported in language processing research (e.g., Inhoff & Rayner, 1986; Rayner & Duffy, 1986). However, readers only experienced higher predictability effects when proofreading for spelling errors that create real words. When spelling errors created nonwords, predictability effects were uninfluenced by the proofreading task. A more recent similar study conducted by Strukelj and Niehorster (2018) showed that reading thoroughly, skimming, and looking for spelling errors all changed eye-movement behaviors in a similar fashion.

Furthermore, many other research areas have shown that task demands can influence real-time and post-interpretive reading behaviors (Binder et al., 2001; Dempsey & Brehm, 2020; Lim & Christianson, 2015; cf. Christianson et al., 2022), making it a worthwhile endeavor to capture a profile of reading behaviors when debugging Python source code as a unique task. Since programmers likely also read source code for comprehension, future work should explore whether reading source code for comprehension versus reading source code for debugging results in different reading behaviors (Chung et al., in prep).

To ground a descriptive analysis of reading while debugging Python source code, it is first important to consider reading behaviors during the reading of natural texts. Schotter and colleagues' (2014) study provides such eye-tracking data for both reading for comprehension and for reading for two kinds of proofreading, which are reported in Table 1. For example, the authors report longer gaze duration times (330-375 msec compared to 240 msec) and a higher probability of rereading (.28-.46 compared to .05) for both proofreading tasks. It is important to note that these data offer us merely a glimpse of other normative datasets of eye-movements across task-specific reading activities, and at no point is it our goal to introduce a statistical comparison between their findings and our own.

Table 1. Mean values of global word-level eye-tracking measures taken from Schotter et al., (2014) by English-speaking participants reading English passages for comprehension. The same measures are given for reading for proofreading of nonwords and reading for proofreading of wrong words, as observed by Schotter et al., (2014). The number of participants in the reading for comprehension group was double since these data were collected from two different experiments with identical stimuli. We average across the two experiments for the reading for comprehension data. Rereading time represents mean time spent rereading only in trials where rereading occurred.

Reading Task	*Eye-Tracking Measure*	*Mean Value*	*SD*
Reading for Comprehension	Skipping Rate	.20	.02
	First Fixation Duration	221	4
	Gaze Duration	240	5
	Rereading Time	290	211
	Rereading Probability	.05	.02
Reading for Proofreading: Nonwords	Skipping Rate	.11	.02
	First Fixation Duration	281	7
	Gaze Duration	375	11
	Rereading Time	347	243
	Rereading Probability	.28	.02
Reading for Proofreading: Wrong Words	Skipping Rate	.11	.01
	First Fixation Duration	264	5
	Gaze Duration	330	8
	Rereading Time	622	544
	Rereading Probability	.46	.03

We are also interested in how eye-movement patterns differ between these types of reading tasks as a function of the type of bug present in the code. For example, when reading

natural texts for comprehension, regressions are often used as an index for processing difficulty (Christianson et al., 2024; Frazier & Rayner, 1982, 1987; Jacob & Felser, 2016; Pickering & Traxler, 1998; Slattery et al., 2013). When readers encounter ungrammatical, infrequent, or unpredictable words and structures, not only will they usually spend more time in first-pass reading, but they will also perform a regressive saccade and reread earlier parts of the sentence. So, the fact that readers spend more time rereading when given a task other than comprehension is suggestive that reading source code for debugging purposes may also lead to higher amounts of rereading. This potential difference is one of several that provide the key motivation behind this study, along with providing a general profile of eye-movements when people are reading source code with different kinds of bugs or none at all.

Current Experiment

The current study may help us understand how cognitive processes differ during source code debugging. Specifically, we seek to address the following research questions:

1. What are the descriptive, non-inferential patterns of global eye-movements associated with reading source code with no errors and how does this differ when there are errors in the source code?
2. How do eye movements reflect the real-time detection of errors in source code comprehension during debugging?

The first research question can be answered by aggregating eye-movement data across participants and trials, thereby producing an aggregate distribution for each measure of interest. Having this descriptive analysis can inform whether or not the same basic eye-movement

patterns are found in reading of natural texts and reading while debugging source code, and, if differences are found, we can observe how they differ. Crucially, we are not actually comparing our descriptive findings to those found in Schotter and colleagues' study or any other study for that matter; rather, the inclusion of those data is meant as a reference to understand general patterns found in studies of natural text reading. The second research question requires inferential models, but it allows us to compare physiological responses to the detection of errors in source code to similar errors in natural texts. This again will allow us to consider commonalities and differences between the two cognitive tasks. Specifically, syntactic errors in reading while debugging source code are expected to elicit immediate processing differences (i.e., first fixation duration, gaze duration), compared to code with no bugs because these reflect surface-level issues like mistakes in spelling or visually-cued syntax (i.e., punctuation and indentations). On the other hand, semantic errors are predicted to elicit later measures of eye-movements like rereading and go-past time because these errors stem from issues with integrating one chunk of code with the larger context of the source code. These patterns follow the assumption that reading between natural texts and source code is mostly similar, which may not prove to be true.

The current study aims to first establish a profile of reading patterns from experienced programmers debugging Python source code. Understanding these baseline patterns can help us better understand how humans change reading strategies to fit a given task as well as provide a reference point for future studies examining psychological processes of debugging source code. Second, we seek to understand how reading patterns differ when encountering semantic and syntactic errors, where a given chunk of source code would either produce an undesired result or produce a runtime error, respectively. To this end, we inspected reading behaviors associated

with the particular bugged region of code compared to that same region in working code as well as general reading behaviors as a function of the code's error type. This latter group of models of global reading behaviors is important because the eye movements associated with debugging may not be directly observable at the time at which the erroneous chunk is initially processed.

In order to examine eye-movement indices while debugging Python source code, experienced programmers were instructed to read through 21 different Python functions to determine whether each function would 1) run without error, 2) run but yield an undesired result, or 3) yield an error message. These latter error conditions were likened to source code versions of semantic and syntactic errors in human text processing, respectively. It is important to note, however, that this is not a perfect comparison. For example, although error messages are often caused by improper syntax, they can also be caused by logical flaws such as incorrect list indexing or acting on the wrong type of variable. It is also not the case that syntactic errors in human text processing result in errors that stop the process (although see Drewnowski & Healy, 1980, and related work for instances where small errors do go by relatively unnoticed); for example, humans are able to overcome impoverished input much better than computers can via good-enough processing strategies (e.g., Christianson, 2016; Ferreira et al., 2002) and noisy-channel updates (Gibson et al., 2013; Levy, 2008; Levy et al., 2009). Nevertheless, evidence from EEG/ERP literature (Kutas & Hillyard, 1984; Osterhout & Holcomb, 1992) establishes potentially differential processes for overcoming syntactic and semantic anomalies in reading (indexed by heightened P600 or N400 effects, respectively), and these differences may also be apparent in the reading of source code.

Another potential difference is that debugging is often performed with the programmer knowing with some confidence that there is something wrong with the code. It is also worth

pointing out that, although debugging for syntactic versus semantic errors may result in different reading behaviors, participants in this experiment have to debug for both error types in addition to the possibility that there is no bug at all. Therefore, the purpose of this study is not to examine differences in strategic reading behaviors depending upon the type of error readers think they are looking for, which would be a lucrative area of research for the future; rather, we are interested in examining eye-movement behaviors associated with debugging in general, with no specific bug type expected *a priori* by readers, as well as how readers of source code react to two types of bugs in source code. For example, syntactic and semantic errors in natural text processing often result in longer first-pass reading times and a higher probability of regressions out of the erroneous region (e.g., Frazier & Rayner, 1982). It is therefore of interest to see whether this profile of reading behavior is also observed upon encountering buggy code.

In this study, we are interested in several commonly studied eye-movement measures of both "early and late" processing. These measures include skipping rate (a binary measure of whether an area of interest was fixated), first fixation duration (the amount of time spent during the first fixation within an area of interest), gaze duration (the amount of time spent fixating an area of interest before leaving that area in either the forward or backward direction), rereading probability (a binary measure of whether an area of interest was fixated after initially leaving that area for the first time), and rereading time (how much time was spent refixating an area of interest). Each of these variables was chosen for a specific cognitive reason (cf. Rayner, 1998; Rayner et al., 2012): skipping rate is predictive of processing depth and can change as a function of text length and complexity (e.g., Slattery & Yates, 2018); first fixation duration and gaze duration are both correlated with early cognitive processes like lexical retrieval, the former being more sensitive to initial visual expectations and the second being more sensitive to linguistic

expectations (Rayner & Reingold, 2015); rereading probability often speaks to the difficulty during later, integrative stages of sentence processing that involve placing words into semantically and syntactically plausible and acceptable roles (e.g., Frazier & Rayner, 1982); and rereading time can often be an indicator of both uncertainty and difficulty in language processing (e.g., Christianson et al., 2023, 2024). These variables were examined in the descriptive analysis reported in the Results section. Additionally, full trial reading was also used as a dependent variable in hierarchical models to determine differences in reading behaviors between different bug types. Bug detection accuracy and self-reported confidence were also used as dependent variables to offer insights into how the different types of errors influence confidence and ability to successfully detect errors.

The eye-tracking measures reported in the current study are hypothesized to reflect similar processes between natural text reading and reading while debugging Python source code. For instance, gaze duration, which is usually reflective of lexical retrieval processes, is hypothesized to reveal processing difficulty in retrieving a chunk of code's meaning, which is sometimes contextually bound (i.e., object labels) and sometimes contextually independent (i.e., words in the basic Python package like "for" or "else"). Thus, one potential difference between bugged and non-bugged AOIs may be found in global gaze duration behaviors such that gaze duration times on average are shorter for bugged code because readers notice an error and then skim the rest of the code to find information relevant to the error. Thus, even early measures of reading behavior may elicit effects based on the bug condition of the source code. This is just one possible finding to illustrate why these measures are worth investigating—as of now, we do not know how these general reading patterns differ as a function of the source code's bug status.

METHOD

Participants

Thirty participants were recruited from the University of Illinois community and received $15 for participation in the experiment. All participants reported having at least two years of experience programming in Python. There was no English language requirement, although all participants were students at the University of Illinois, so a certain level of English proficiency was assumed. Despite the lack of language requirement, only one student did not self-report as a native English speaker. Participant demographic and relevant coding experience data are reported in Table 2. There are a few patterns worth mentioning from the demographic data. First, in terms of programming experience, most participants indicated having experience with at least one additional programming language, and the variance in number of years of Python experience was not very large (M = 3.2 years, SD = 2.1 years), meaning this study is measuring eye-movement patterns of a rather specific range of programmers with about 2 to 5 years of Python experience. The race and ethnicity of participants is representative of several groups (Table 2); however, it is not exhaustive. Moreover, the majority of participants in this study were male, which, although closely resembling some estimates of demographic breakdown in programming disciplines (7.51% Female [GitLab, 2020]), makes it difficult for the findings to generalize to diverse populations. Although we assume demographic differences to be minimal in their influence on eye-movement behaviors, we acknowledge that the current sample is not fully representative, and future work should aim to provide a more inclusive sample.

Table 2. Demographic breakdown of study participants. Participants were told they must have at least two years of experience programming in Python to participate. Python comfort was assessed via a five-point Likert scale with 1 representing not comfortable at all and 5 representing completely comfortable. All other variables were open answer except for Language, which was a binary choice of self-identification as a native speaker of English or not.

Participant variable	**Breakdown**
Age	*M* = 21.7 *SD* = 4.2
Gender	27 Males 3 Females
Race/ethnicity	12 South/Southeast Asian 8 Asian/East Asian 7 White 3 Hispanic
Major field of study	18 Computer Science/Engineering 5 Engineering 3 Statistics/Math 3 Physics/Biophysics 1 Psychology
Language	29 Native English 1 ESL
Programming experience	*M* = 4.8 Years *SD* = 2.1 Years
Python experience	*M* = 3.2 Years *SD* = 1.5 Years
Python comfort	*M* = 4.1 *SD* = 0.7

Materials

When designing the Python functions for this study, the goal was to include a diverse set of functions with various bugs to understand general reading processes that are not constrained by any particular type of function or bugged code. For that reason, we developed 21 Python functions and created 3 versions of each function: no bug, syntactic bug, semantic bug. Syntactic bugs were defined as bugs that would produce a run-time error, whereas semantic bugs would produce an undesired result. This maps onto syntactic and semantic errors in natural text reading in several ways, but is not a perfect match. Specifically, syntactic errors in source code stem

from lower-level information like improper character usage or visual syntactic cues (e.g., improper indentations, spelling mistakes, incorrect punctuation). Thus, syntactic anomalies in source code may be detected from earlier, lower-level visual processing stages compared to syntactic errors in natural text that require integration across the sentence (e.g., subject–verb agreement). However, these errors do map between different text types in that syntactic errors in regular texts also do not usually require as much contextual support to detect (e.g., morphological errors or issues with word order in English) compared with semantic errors. Semantic errors, on the other hand, often lead to conflict in terms of plausibility, which requires evaluating the error with the greater context of the sentence and real-world expectations (e.g., interpreting “Hannah mowed the giraffe” requires understanding that giraffes are not usually mowed).

Areas of interest (AOIs) were constructed by considering parts of code that may be processed as a chunk, much like how a word may be processed as a unit. For example, Figure 2 shows how AOIs are constructed across three different versions of the same item. Importantly, the bugged versions of items never changed these function properties. One item did not meet this particular criterion after inspection following the conclusion of the experiment, and so it was discarded prior to analysis. These AOIs differ particularly in terms of their length and their complexity, defined in the current study as the number of characters and the number of embedded arguments, respectively. Table 3 reports various statistical properties of the different items used in this experiment. For each bug type, there were three different sub-types. For syntactic conditions, bugs could be of three different types: incorrect indentation or punctuation, omission of some required character(s), or spelling and terminology errors. For semantic

conditions that yielded undesired results, bugs could be of three different types as well: incorrect indexing, incorrect mathematical calculation or operator, or a wrong variable being used.

Figure 2. An example item with all three versions: no bug, semantic bug, and syntactic bug. Because all content of the code is in some area of interest, we use the "\" characters to delineate the areas of interest. These were not visible to participants. Participants saw the source code with syntax highlighting (i.e., keywords, functions, comments, etc. shown in different colors to delineate purpose), as is typical in source code editors.

```
def is_palindrome\(word):\
    #This function determines whether a word is the same spelled backwards (i.e. a palindrome).\
    #Example: Input: 'racecar'\
    #         Output: True\
    order =\ []\
    rev_order =\ []\
    for i in \range(len(word)):\
        order.append(word[i])\
        rev_order.append(word[-(i+1)])\
    if order ==\ rev_order:\
        return True\
    else:\
        return False\

#Bug: Semantic: -(i) should be -(i+1)
def is_palindrome\(word):\
    #This function determines whether a word is the same spelled backwards (i.e. a palindrome).\
    #Example: Input: 'racecar'\
    #         Output: True\
    order =\ []\
    rev_order =\ []\
    for i in \range(len(word)):\
        order.append(word[i])\
        rev_order.append(word[-(i)])\
    if order ==\ rev_order:\
        return True\
    else:\
        return False\

#Bug: Syntactic: index uses () instead of []
def is_palindrome\(word):\
    #This function determines whether a word is the same spelled backwards (i.e. a palindrome).\
    #Example: Input: 'racecar'\
    #         Output: True\
    order =\ []\
    rev_order =\ []\
    for i in \range(len(word)):\
        order.append(word(i))\
        rev_order.append(word[-(i+1)])\
    if order ==\ rev_order:\
        return True\
    else:\
        return False\
```

Table 3. Descriptive summary of item and area of interest properties. The area of interest types are defined as follows: *Variable* includes chunks of code that create variables within the function; *Logic* includes conditional statements like "if"; *Instructions* includes all lines commented out of the code with "#"; *Loop* includes all lines that initiate an iterative loop; *Definition & Import* includes all function definition and import lines; *Return* includes lines with a return function; *Function* includes any line not previously categorized that uses a function (e.g., "print()").

Item variable	**BREAKDOWN**
Number of lines	*M* = 14.8 Lines *SD* = 3.0 Lines
Number of area of interests	*M* = 23.7 AOIs *SD* = 6.1 AOIs
Average area of interest complexity	*M* = 1.2 Levels *SD* = 0.1 Levels
Average area of interest length	*M* = 3.2 Characters *SD* = 0.7 Characters
Distribution of AOI types	Variable = 42% Logic = 15% Instructions = 14% Loop = 10% Definition & Import = 9% Return = 7% Function = 3%

All items contained at least three lines of comments directly underneath the definition line. The first line (or multiple lines, depending on the complexity of the function's intended use) explained the desired goal of the function. The last two lines were always an example input and example output. AOIs were coded into one of several different categories depending on their purpose (e.g., "Variable" if the AOI's purpose was to create a variable within the function). Some items imported packages before the definition line, but no information about those specific packages was given. In general, there was always the chance that participants may not be familiar with a given function that appeared in the code, although only functions and packages from the standard library of Python were included to improve the chances that participants would be familiar with them. To better inform the data patterns we collected, all items included

confidence ratings immediately following the bug detection prompt (see Procedure section). Lastly, the items retained typical syntax highlighting (i.e., color coding) from the source code editor they were created in; however, in many editors, syntactic bugs result in color changes to cue programmers that some line of code is erroneous. Therefore, we retained the color that the bugged AOI would appear in assuming there was no error. This decision was necessary to prevent participants from immediately using text color information to find bugs while still retaining visual color cues that experienced programmers use on a daily basis. Participants were warned of this before starting the experiment.

Procedure

Participants first signed consent and completed a written questionnaire about their Python and programming experience. Next, they were read instructions followed by three practice items showcasing a trial with no bug, a trial with a syntactic bug, and a trial with a semantic bug. For each trial, participants first saw a screen reminding them of the three bug categories an item could appear in. Participants were instructed to use a button to proceed to the next screen that contained the item's code, where they were instructed to read the code silently until they determined whether there was no bug, a semantic bug, or a syntactic bug. After this determination, participants were instructed to proceed with the button to the next screen, where they would indicate their answer. Immediately following their response, participants indicated whether they were very confident, somewhat confident, or not at all confident in their response. After indicating their confidence level, the next trial would begin. Participants were told three additional pieces of information about the task: 1) they should not rely solely on the example input and output for determining if a function would work, 2) the color scheme is not designed to

aid in debugging (see previous section), and 3) they should try their best to figure out what unfamiliar functions do based on context. Participants did not receive feedback about their responses. About halfway through the experiment, participants were given a break no longer than two minutes to rest their eyes. At the end of the experiment, participants filled out a demographic survey before receiving compensation.

After participants completed the practice items, they were calibrated using a 9-point calibration procedure on the desktop SR Research EyeLink 1000+ eye-tracker system[2]. Calibration was deemed acceptable if validation showed less than .1° divergence from the initial calibration. Viewing was binocular, but only the left eye was tracked. For some participants, the right eye was instead used due to calibration difficulties. Text was presented in 14-pt Courier New monospace font on a 20-inch LCD monitor with a 120 Hz refresh rate, and areas of interest bordered this text with an additional buffer of blank space extending about one character above and below. The monitor was 60 cm away from participants' eyes, which corresponds to about 1° of visual angle spanning approximately 3.5 characters. Eye-movements were recorded with a sampling rate of 1000Hz and spatial resolution of 0.01°. After the break, participants were calibrated again, and calibration was sometimes necessary if the tracker lost track of their eyes. The task took on average 30 minutes to complete.

Data Availability Statement

All data and analysis code are available online at https://osf.io/49wga/.

[2] For eye tracker specifications, see SR Research, ExperimentBuilder 2.4.77 User Manual. The default event detection algorithm used by SR Research is an Identification by Velocity Threshold (IVT) algorithm. For cognitive experiments, the velocity threshold is set to 30 degrees/sec by default.

Results

Data Cleaning Procedures

No participants were removed based on qualifying criteria or inattentiveness. Each trial from each participant was inspected in SR Research DataViewer software. Minimal manual edits were made for fixations that systematically appeared slightly above or below areas of interest. Fixations that appeared outside any area of interest and did not belong to any systematic pattern were removed prior to analysis. Next, remaining fixations that lasted less than 80 msec were removed from the dataset, resulting in a loss of less than 5% of fixations. These data cleaning guidelines were meant to be less strict than is commonly used in psycholinguistic studies of reading, since we did not have *a priori* expectations for durational distributions. Following this logic, no participants were removed prior to analysis due to low accuracy since these bugs were not necessarily easy to find, even for experienced programmers. Furthermore, although we treat accuracy as binary (i.e., correct or incorrect) for our analyses, the chance rate for accurate responses was 33%, not 50%. Lastly, outliers were not removed from the dataset because we had no *a priori* expectation beyond the aforementioned parameters of how the distributions should look. In other words, we wanted to be as conservative as possible with the data because our prior knowledge was not sufficient to make a decision regarding outlier cutoffs, and one of the main goals of the paper was to illustrate what those distributions looked like.

Descriptive Analysis of Reading Measures

Means and standard deviations for these measures by experimental condition are reported in Table 4 below. AOIs that contained instructions were not included since these items were

essentially read for comprehension—the computer does not read them. Although AOIs in the current study were constructed to isolate chunks that represented conceptual units, there is likely not a one-to-one correspondence here with word-level reading patterns, like in Schotter et al. (2014). However, there are still many parallels. For example, oculomotor function should behave similarly since the eyes need to move in the same fashion when reading any kind of character-based information. Second, although the task is different in reading natural vs. programming languages, we are assuming the same general mechanism of early eye-movement behaviors during both types of reading as a function of text predictability, frequency, and length, as described for example in the E-Z Reader model (Reichle et al., 2003).

Table 4. Means and standard deviations of globally averaged reading behaviors by experimental condition while reading for debugging Python source code, rounded to nearest msec for continuous measures and nearest hundredth for probabilistic measures.

Reading measure	***No Bug Mean (SD)***	***Syn Bug Mean (SD)***	***Sem Bug Mean (SD)***
First fixation duration	228 (97)	230 (101)	228 (94)
Gaze duration	387 (524)	396 (541)	378 (487)
Rereading time	2392 (3305)	1949 (2679)	2066 (2869)
Rereading probability	.76 (43)	.74 (.44)	.74 (.44)
Skipping probability	.53 (.50)	.54 (.50)	.53 (.50)

Although mean first fixation duration while debugging Python source code was close to what we might see for reading for comprehension as reported by Schotter et al. (2014) (Table 1), its standard deviation, like all of the variables reported in the current study, was much larger. On the other hand, mean gaze duration while debugging Python source code was substantially longer

than that of either type of proofreading for natural texts. The same was true for rereading time, suggesting participants spent substantially more time rereading while debugging Python source code compared to either type of proofreading or reading for comprehension of natural texts. Additionally, there was a higher probability of rereading compared to natural texts (72% compared to 5% [reading for comprehension], 28% [reading for proofreading spelling errors], and 46% [reading for proofreading nonwords]). Participants skipped nearly half the AOIs in this experiment on average compared to about 20% during reading for comprehension and 10% for either type of proofreading of natural texts. These findings are discussed further in the General Discussion.

Global Reading Models

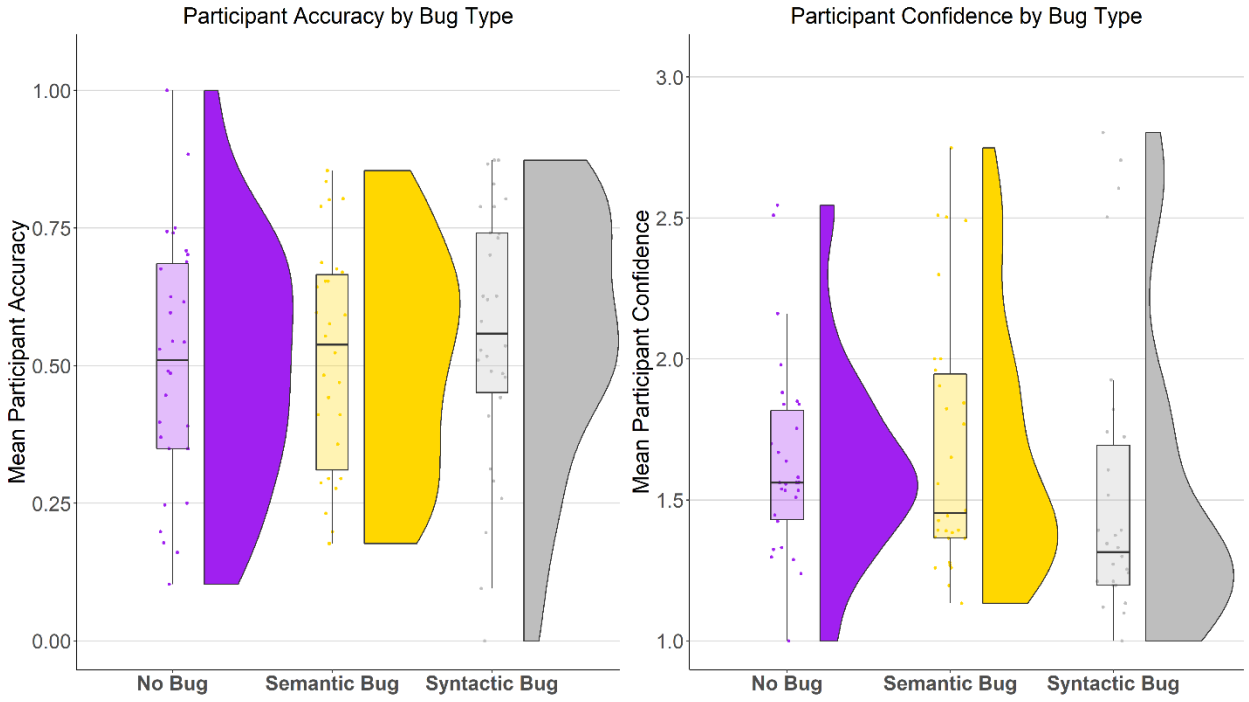
To assess whether there were significant differences in these eye-movement measures between bug conditions, we fit a series of Bayesian hierarchical models (also known as mixed effects models or nested models) to the data using the *brms* package (Bürkner, 2017) in R version 4.0.3 (R Core Team). For each model with a continuous dependent variable, log-transformed data were regressed onto a fixed effect of condition (treatment coded with no-bug baseline). Since there were three contrasts of interest, two separate models were run for each dependent measure with either the no-bug condition or the syntactic bug condition as the baseline. The choice to use this contrast scheme may cause issues with multiple comparisons (but see Gelman et al., 2012 for why Bayesian models are more robust to these issues); however, Helmert contrasts were not deemed suitable for the current analysis because there was no *a priori* expectation for a baseline against which both treatment conditions would similarly compare. Nevertheless, this modeling approach may still lead to an inflated Type I error rate;

however, we find no concerning contradictions between models, suggesting this is not the case. For instance, comparing the syntactic and semantic bugs with each other and then comparing their mean with a no bug baseline would rely on the assumption that we expect the directionality of differences between each treatment group and the no-bug condition to be the same. For this reason, we report the output of two models with different treatment contrasts. An additional control of Python experience (in years) was entered as a fixed effect as well, and number of AOIs was entered as a length control for the full trial reading time model. Similarly, chunk count, or the number of word-like chunks in a given AOI, was entered as a fixed effect for skipping and regression-in rate since these dependent measures were not aggregated to maintain their binary distribution. These control fixed effects are not reported in tables but are in the supplemental code. Random intercepts were entered by Participant and by Item and maximal random slopes were included in all models. All models with continuous dependent measures were fit to a Gaussian distribution aggregated across AOIs within each item. This means that each measure was aggregated such that each individual trial of an item by a specific participant produced one mean value. This approach was taken to analyze general behaviors when reading for debugging and avoid by-trial influences, further controlled for in our random effects structure. Models with binary dependent measures were instead fitted with models using a Bernoulli distribution.

Models were run with mildly informative priors that allow for a wide variation of mean reading behaviors as well as effect sizes, and these are reported in all model output tables. Models were run for 7500 iterations, 2500 of which were warmup, with 4 chains. To interpret models, we used 95% credible intervals (CrIs) as a heuristic cutoff for identifying effects that may be meaningful. If a 95% CrI does not contain 0, that indicates that at least 97.5% of

posterior estimates showed an effect in that given direction. Accuracy and confidence distributions by participant and by bug type are illustrated in Figure 3 below. First fixation duration, gaze duration, rereading time, and full trial reading time were all fit using a Gaussian distribution aggregated across AOIs within each trial, whereas skipping rate and regression in probability were fit using a Bernoulli distribution to unaggregated AOI data. For the Gaussian models, priors included an intercept prior [normal(0,10)], beta prior [normal(0,1)], and group-level standard deviation prior [normal(0,1)]. For the Bernoulli models, including accuracy and confidence models, the intercept prior was changed [normal(0,2)] for sampling efficiency.

Figure 3. Mean accuracy and confidence by participant across experimental conditions. Raincloud plots throughout the paper adapted from Allen et al., 2019. Individual points represent mean values, violin plots show the shape of the distribution, and the boxplots represent the median and 25% to 75% quartiles.



Model outputs for global reading measures are reported in Table 5 below, accuracy and confidence model outputs are reported in Table 6, and distributions of select measures are reported in Figures 4, 5, and 6 below. First, experimental condition seemed to have no effect on accuracy rate, whereas syntactic conditions elicited lower levels of confidence overall compared to both no-bug and semantic-bug conditions. The eye-movement models revealed simple effects between syntactic bug and no-bug conditions such that syntactic bugs led to a lower probability of rereading, less time spent rereading, and less time spent reading the full trial. The only difference between no bug and semantic bug conditions was that the latter resulted in slightly less overall rereading, just like the syntactic bugs did. These results suggest that no-bug conditions led to more rereading than syntactic trials, which is likely due to the uncertainty involved in deciding no bug exists versus finding a bug, particularly when that bug is syntactic in nature. This is seemingly in contrast with the finding that confidence is higher in no-bug conditions versus syntactic bug conditions; however, it could be that the additional time spent rereading the no-bug trials led to greater confidence in answers. Finally, higher rates of skipping and lower rates of regressions-in were observed for AOIs with fewer chunks, similar to length effects found during reading of natural texts, and longer trials unsurprisingly led to longer overall reading times.

Tables 5. Model outputs for global reading behaviors. Simple effects for all comparisons were obtained by running two identical models, the first with a no bug baseline and the second with a syntactic bug baseline. Bolding denotes effects where the 95% CrI does not cross 0.

Dependent Measure	*Baseline*	*Effect*	*Estimate*	*SE*	*95% CrI*
First Fixation Duration	No Bug	Semantic Condition	.01	.01	[-.01, .04]
	No Bug	Syntactic Condition	.01	.01	[-.02, .04]
	No Bug	Python Experience	-.01	.02	[-.04, .02]
	Syntactic	No-bug Condition	-.01	.01	[-.03, .02]
	Syntactic	Semantic Condition	.01	.01	[-.02, .03]
	Syntactic	Python Experience	-.01	.02	[-.05, .02]
Gaze Duration	No Bug	Semantic Condition	-.01	.03	[-.07, .06]
	No Bug	Syntactic Condition	.03	.04	[-.04, .11]
	No Bug	Python Experience	.22	.32	[-.43, .84]
	Syntactic	No-bug Condition	-.04	.04	[-.11, .04]
	Syntactic	Semantic Condition	-.04	.03	[-.11, .02]
	Syntactic	Python Experience	.21	.31	[-.47, .80]
Skipping Rate	No Bug	Semantic Condition	-.01	.07	[-.14, .12]
	No Bug	Syntactic Condition	.02	.06	[-.10, .13]
	No Bug	Python Experience	.03	.06	[-.08, .14]
	No Bug	**Chunk Count**	**-.35**	**.01**	**[-.38, -.32]**
	Syntactic	No-bug Condition	-.02	.06	[-.15, .10]
	Syntactic	Semantic Condition	-.03	.06	[-.15, .10]
	Syntactic	Python Experience	.03	.06	[-.08, .14]
	Syntactic	**Chunk Count**	**-.35**	**.01**	**[-.38, -.32]**

Regression In Prob.	No Bug	Semantic Condition	-.08	.07	[-.21, .05]
	No Bug	**Syntactic Condition**	**-.14**	**.07**	**[-.28, .00]**
	No Bug	Python Experience	-.02	.05	[-.13, .06]
	No Bug	**Chunk Count**	**.07**	**.01**	**[.05, .10]**
	Syntactic	**No-bug Condition**	**.15**	**.07**	**[.01, .29]**
	Syntactic	Semantic Condition	.06	.06	[-.07 ,.20]
	Syntactic	Python Experience	-.03	.04	[-.12, .06]
	Syntactic	**Chunk Count**	**.07**	**.01**	**[.05, .10]**
Rereading Time	No Bug	Semantic Condition	-.17	.10	[-.38, .04]
	No Bug	**Syntactic Condition**	**-.28**	**.12**	**[-.51, -.05]**
	No Bug	Python Experience	-.02	.09	[-.22, .15]
	Syntactic	**No-bug Condition**	**.28**	**.11**	**[.07, .49]**
	Syntactic	Semantic Condition	.11	.12	[-.15, .35]
	Syntactic	Python Experience	-.02	.09	[-.21, .15]
Full Trial Reading Time	**No Bug**	**Semantic Condition**	**-.12**	**.06**	**[-.25, .00]**
	No Bug	**Syntactic Condition**	**-.25**	**.08**	**[-.40, -.10]**
	No Bug	Python Experience	.03	.06	[-.17, .09]
	No Bug	**Number of AOIs**	**.03**	**.01**	**[.01, .05]**
	Syntactic	**No-bug Condition**	**.25**	**.07**	**[.11, .39]**
	Syntactic	Semantic Condition	.12	.08	[-.04, .29]
	Syntactic	Python Experience	-.04	.06	[-.17, .08]
	Syntactic	**Number of AOIs**	**.03**	**.01**	**[.01, .05]**

Table 6. Model outputs for accuracy and confidence. Simple effects for all comparison were obtained by running two identical models, the first with a no bug baseline and the second with a syntactic bug baseline. Bolding denotes effects where the 95% CrI does not cross 0.

Dependent Measure	*Baseline*	*Effect*	*Estimate*	*SE*	*95% CrI*
Accuracy	No Bug	Semantic Condition	.09	.29	[-.48, .67]
	No Bug	Syntactic Condition	.16	.40	[-.63, .96]
	No Bug	Python Experience	.16	.12	[-.06, .42]
	Syntactic	No-bug Condition	-.13	.36	[-.84, .59]
	Syntactic	Semantic Condition	-.03	.38	[-.79, .73]
	Syntactic	Python Experience	.18	.12	[-.04, .43]
Confidence	No Bug	Semantic Condition	.02	.08	[-.13, .17]
	No Bug	**Syntactic Condition**	**-.15**	**.07**	**[-.30, .00]**
	No Bug	Python Experience	-.07	.05	[-.17, .04]
	Syntactic	No-bug Condition	**.15**	**.09**	**[-.03, .32]**
	Syntactic	**Semantic Condition**	**.16**	**.07**	**[.03, .29]**
	Syntactic	Python Experience	-.07	.05	[-.18, .04]

Figure 4. Mean rereading probability by participant across experimental conditions.

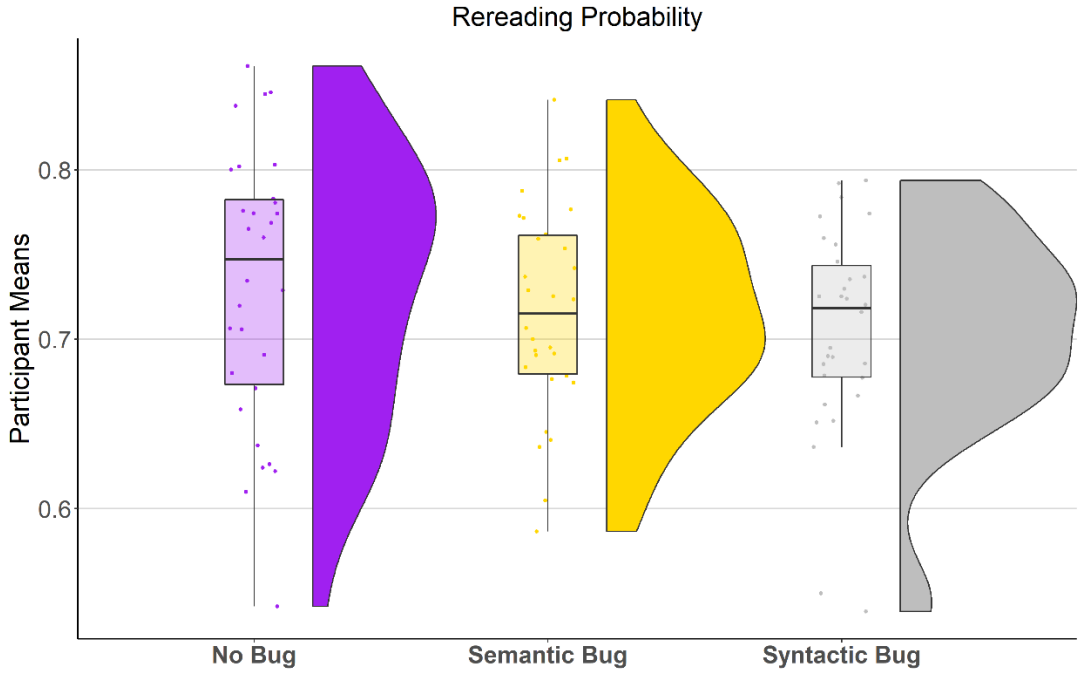


Figure 5. Mean log-transformed rereading time by participant across experimental conditions.

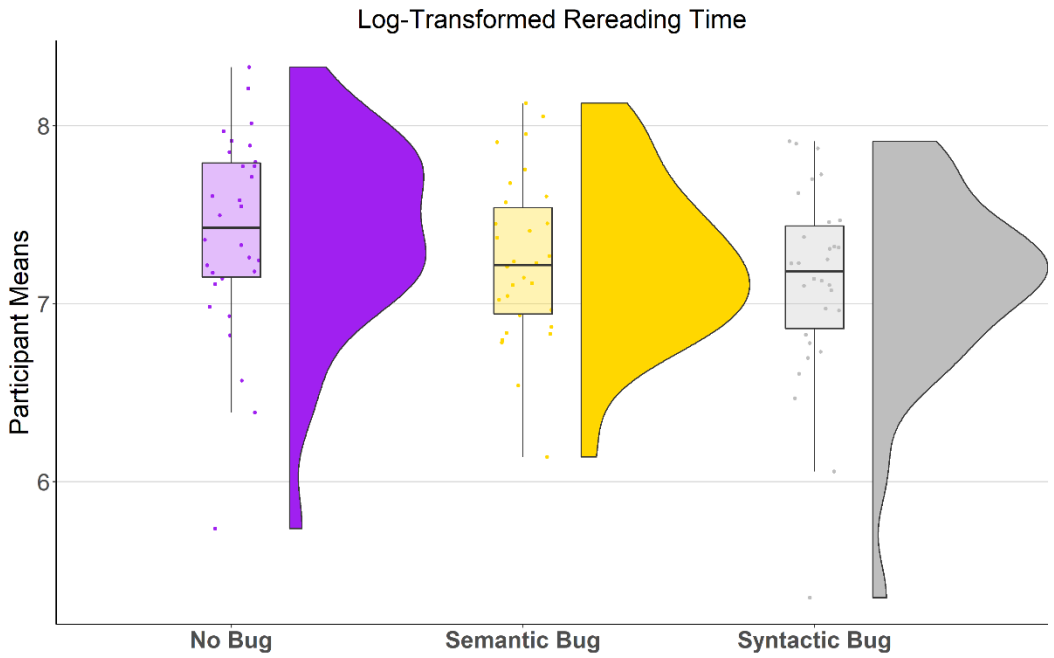


Figure 6. Mean log-transformed full trial reading time by participant across experimental conditions.

Bug Identification Models

To investigate the profile of eye movements during the reading of bugged source code, we fit Bayesian hierarchical models to the target AOI (the bug) for all items. Model outputs are reported in Table 7. Since the syntactic bug and semantic bug often occurred on different lines from one another, two sets of analyses investigated how each type of bug differed from the reading of the same AOI with no bug. This allowed us to use sum contrasts instead of treatment contrasts (no-bug condition = -.5, syntactic/semantic condition = .5). We also entered length in non-punctuation characters and the number of embedded levels within an AOI as fixed effects, referred to henceforth as AOI length and AOI complexity respectively, and we used the same random intercepts by Participant and by Item as in the other models. Since binomial responses

were not averaged like for the global models, we fit models with binomial response dependent variables to a Bernoulli distribution and used slightly adjusted mildly informative priors accordingly (i.e., we changed the *SD* of the intercept parameter from 10 to 2 for more efficient sampling). From these models, the only text characteristic effect found was a main effect of AOI Complexity such that higher complexity predicted a higher probability of regressing out of the AOI. In addition, a main effect of condition was found in gaze duration times such that semantic conditions elicited longer gaze duration times compared with no-bug conditions. This suggests that semantic information is processed to some degree during early stages of processing.

Table 7. Model outputs for reading behaviors of bugged AOIs. Main effects for all comparison were obtained by running two identical models, the first including only no bug and semantic bug trials and the second including only no bug and syntactic bug trials. Bolding denotes effects where the 95% CrI does not cross 0.

Dependent Measure	*No-bug Contrast*	*Effect*	*Estimate*	*SE*	*95% CrI*
First Fixation Duration	Semantic	Condition	.03	.06	[-.09, .16]
	Semantic	Python Experience	.01	.02	[-.03, .05]
	Semantic	Chunk Count	.02	.03	[-.03, .08]
	Semantic	Complexity	-.09	.08	[-.26, .08]
	Syntactic	Condition	.01	.05	[-.08, .10]
	Syntactic	Python Experience	-.02	.02	[-.07, .03]
	Syntactic	Chunk Count	.00	.01	[-.02, .03]
	Syntactic	Complexity	-.02	.04	[-.11, .06]
Gaze Duration	**Semantic**	**Condition**	**.23**	**.10**	**[.03, .42]**
	Semantic	Python Experience	-.03	.05	[-.12, .06]
	Semantic	**Chunk Count**	**.20**	**.05**	**[.09, .30]**
	Semantic	Complexity	.05	.15	[-.25, .36]
	Syntactic	Condition	.09	.10	[-.11, .29]
	Syntactic	Python Experience	-.01	.05	[-.10, .08]
	Syntactic	Chunk Count	.04	.04	[-.03, .12]
	Syntactic	Complexity	.12	.12	[-.11, .35]
Skipping Rate	Semantic	Condition	.10	.34	[-.56, .76]
	Semantic	Python Experience	.00	.17	[-.34, .33]
	Semantic	**Chunk Count**	**-.58**	**.24**	**[-1.07, -.14]**
	Semantic	Complexity	-.08	.56	[-1.17, 1.03]

	Syntactic	Condition	-.15	.33	[-.80, .51]
	Syntactic	Python Experience	.04	.16	[-.28, .35]
	Syntactic	Chunk Count	-.37	.22	[-.84, .02]
	Syntactic	Complexity	-.70	.49	[-1.72, .24]
Regression Out Prob.	Semantic	Condition	.09	.32	[-.54, .73]
	Semantic	Python Experience	.01	.16	[-.32, .34]
	Semantic	Chunk Count	.28	.25	[-.22, .76]
	Semantic	Complexity	-.24	.66	[-1.51, 1.08]
	Syntactic	Condition	.11	.32	[-.53, .74]
	Syntactic	Python Experience	-.12	.13	[-.37, .12]
	Syntactic	Chunk Count	.20	.17	[-.11, .57]
	Syntactic	**Complexity**	**1.09**	**.45**	**[.22, 1.98]**
Rereading Time	Semantic	Condition	.16	.15	[-.13, .45]
	Semantic	Python Experience	-.01	.08	[-.17, .15]
	Semantic	**Chunk Count**	**.19**	**.10**	**[.00, .38]**
	Semantic	Complexity	.45	.29	[-.15, 1.00]
	Syntactic	Condition	-.12	.17	[-.47, .22]
	Syntactic	Python Experience	.01	.08	[-.15, .17]
	Syntactic	Chunk Count	.07	.05	[-.04, .18]
	Syntactic	Complexity	.21	.18	[-.13, .58]

GENERAL DISCUSSION

The goals of the current study included first establishing eye-movement indices of reading while debugging Python source code before examining reading patterns associated with different types of bugs. This section focuses on each of these goals in turn before putting forth a proposal that psycholinguistic analysis can and should inform advances in our understanding of source code reading. Finally, a few outstanding questions are highlighted to showcase the directions we believe are most critical for developing educational interventions and real-time strategies for readers of source code at all levels of expertise.

Eye-Movement Indices of Reading for Debugging Python Source Code

The global reading patterns observed in the current experiment, reported in Table 4, illustrate a descriptive pattern that appears distinct from the reading of natural texts both for comprehension and for different types of proofreading. In turn, first fixation duration, gaze duration, rereading time, rereading probability, and skipping probability together form a unique index of eye movements during the reading for debugging of Python source code. First fixation duration seems to be in line with reading natural texts, perhaps most closely with reading natural texts for comprehension. This perhaps makes sense since the longer first fixation durations for reading natural texts for proofreading likely reflect a lower-level, bottom-up strategy that would be beneficial for detecting spelling errors, whereas the kind of troubleshooting in Python source code reading was necessarily vaguer in terms of the source of the bug. For example, there could be a misspelled word; however, there could also be incorrect logic, incorrect indentation levels, the wrong variable called, etc., which would all require higher-level, top-down reasoning to

detect. Therefore, it follows that first fixation durations, an early measure of processing more indicative of visual, graphemic, or lexical processing, would be closer to that of reading natural texts for comprehension.

Global gaze duration times show a clear pattern, where reading for debugging Python source code elicits longer gaze durations than reading natural texts for both comprehension and proofreading. A potential caveat could be that the length of the AOIs is modulating this difference; however, the mean AOI length is only 3.2 characters, which does not differ meaningfully from the average length of English words in most texts, for example 4.75 characters in the Brown corpus (Francis & Kucera, 1979; von der Malsburg, 2022). It is therefore more likely that, although reading source code does not differ in terms of initial fixation duration, the relatively lower regularity of its form compared to natural texts may necessitate a higher number of fixations (e.g., Stites et al., 2013).

The differences grow larger when we look at rereading patterns. For example, areas of interest are reread on average for one full second longer than reading for proofreading of natural texts from Schotter et al.'s study, and rereading is 1.5 times more likely for reading for debugging Python source code. Reading for debugging Python source code also elicits more skipping. The increased rate of rereading may reflect a greater need for referencing earlier parts of the input than is present in reading natural language texts. The increased rate of skipping, on the other hand, might show that source code reading for debugging does not require as strictly an incremental processing strategy as reading natural texts for comprehension or for troubleshooting. For instance, experienced programmers may go through the function in a mostly vertical manner first to understand the global structure of the function (Busjahn et al., 2015; Turner et al., 2014), whereas such a strategy would not prove as fruitful in natural text reading. It

is also possible that, like in natural reading, reading source code utilizes parafoveal processing to inform skipping behaviors (see Schotter et al., 2012 for a review on parafoveal processing in reading natural texts). For example, when reading a variable assignment line such as "my_list = []", experienced programmers may process the empty brackets in their parafoveal view and then decide to skip to the next line. However, such parafoveal preview effects require future studies to confirm.

Lastly, it is important to note that these comparisons are not backed by statistical models; that is, we cannot say with certainty that these differences are significant or reliable. However, we argue that, by providing a profile of reading behaviors or a specific task much like Schotter et al. (2014) did, we are allowing for observational patterns to be assessed. Nevertheless, future work should focus on directly comparing these reading contexts within an experimental design to more concretely assess the degree of these differences.

Bug-Type Differences in Global Reading Measures

The only set of findings from the models of global reading behaviors linked to specific bugs indicated that syntactic bugs elicited less rereading than the no-bug condition. Since no difference was found between the no-bug condition and semantic bug condition, it is likely that this finding stems from a lack of rereading in syntactic bug conditions specifically. One reason why this is likely the case is that syntactic bugs, once perceived, do not require integration into the broader context of the code to determine whether they are truly bugs or not. For example, if a function word is misspelled, then you do not need to understand what that line of code was trying to do to know that it would produce a runtime error. Interestingly, syntactic bugs led to lower

confidence despite a lack of rereading, which on the surface may seem contradictory since rereading is often an index of uncertainty; however, rereading in this particular case would not help participants since syntactic errors often hinge on one single, relatively small cue. This is perhaps why confidence is low for syntactic bugs even though there is not much rereading. The lack of effect between semantic and no-bug conditions was also unexpected, but this is perhaps driven by the fact that both conditions require a thorough reading of the code to understand whether it will produce the desired output or not.

Bug Detection Reading Patterns

When looking at how reading patterns of buggy AOIs differed between conditions, the first takeaway is that complexity of the AOI influenced the probability of rereading just as is reported in natural text reading where the complexity of words influences rereading behavior. Interestingly, the only effects of bug type found were between semantic bug trials and no-bug trials such that bugged AOIs in semantic trials elicited longer gaze durations. It seems, then, that semantic bugs may influence early, first-pass measures of reading. When compared with semantic and syntactic error processing while reading natural texts, this is the opposite of what would be expected. This finding is perhaps indicative of the heavier reliance on top-down strategies when reading source code compared to natural texts, as proposed in previous research (Brooks, 1983; Schneiderman & Mayer, 1979). Bugs such as erroneous indexing, calling the wrong variable, or incorrect mathematical operations may interrupt logical predictions made by experienced programmers. This would be consistent with the large extant body of psycholinguistic literature showing that readers can make predictions in constrained contexts that cause processing disruptions if they do not match the eventual input (e.g., Federmeier, 2007;

Kuperberg & Jaeger, 2016). When coupled with recent research showing that programming language is more predictable than written language, it seems a likely case that experienced programmers learn to rely more on semantic information at earlier stages of processing than is typical in natural text reading; however, more work is needed to determine the interplay of predictability and bug detection-related behaviors.

Limitations and Future Directions

Perhaps the most obvious limitation of the study is that we cannot speak to the generalizability of these Python source code findings to reading while debugging other programming languages. Future work is needed to determine if such differences between languages exist and to what extent they differ, if at all. Moreover, although we refer to the source code reading activity in the current study as debugging, debugging in the real world is different in several ways. First, the color coding can be used as a cue to syntax errors in most source code editors, whereas we did not allow for this to ensure the errors required a somewhat thorough reading of the code. Second, participants were not able to actually run the code to see either an inappropriate output or a runtime error, both of which could help programmers find the particular issue if there is one. Third, functions only represent a part of a programming language and therefore do not represent all types of materials that are being read and debugged by experienced programmers. Although these discrepancies were necessary in the current study for the sake of limiting variability between items and avoiding certain caveats, future work is needed to ascertain how these conditions contribute to eye-movements during debugging of source code.

Another limitation of the current study is that even experienced programmers may not know whether a bug they find will yield an error or simply produce the wrong result, especially when they are not given the option of running the code to find out. Although the models reported in this manuscript treat accuracy as a binary, participants reported a semantic bug in 30% of inaccurate trials for syntactic bug conditions and reported a syntactic bug in 45% of inaccurate trials for semantic bug conditions. Since the current experiment had participants decide between three options, it makes the most sense to treat accuracy as accurate only if the correct choice was selected. However, to investigate whether participants were better at detecting a bug in general versus determining the type of bug, we reran the accuracy model with this more general version of accuracy and found simple effects such that both types of bug condition elicited higher accuracy than the no-bug condition. This can be found in the supplemental code. That is, participants were more successful at detecting a bug than determining that there was no bug, although their ability to determine whether the bug was syntactic or semantic was less successful. Moreover, participants were significantly better at determining that a bug was semantic in nature than determining that it was syntactic (i.e., would produce a runtime error). This may also be due to the fact that there was no way for participants to check if the code would run in the current experimental design. One might argue that this lack of ability to discern between bug types means that the manipulation was not salient enough for these programmers; however, the ability to explicitly classify types of errors and the ability to identify errors in general are different cognitive skills. One potential avenue for future research would be to limit the type of bugs to either semantic or syntactic. Since having participants decide between bugs creates additional noise, which is arguably more generalizable to the debugging process in the

real world, it still may obfuscate how patterns of reading influence successful detection of particular bugs.

Although we chose the data reported by Schotter et al. (2014) as an example of a similar psycholinguistic dataset of task-specific eye-movements, we did not make any sort of statistical comparison between our data and theirs, and no single study should suffice to capture the variability that goes into the process of reading. Thus, future studies are needed to extend those reported in the current manuscript. Additionally, as has been found in reading for natural texts and for reading of source code, the reading strategies and processes used by programmers are likely different depending on level of experience, and there are likely differences within a given reader with how they read and proofread text and how they read and debug code. The current sample likely represents an intermediate population who are familiar with code but have spent much less than a decade coding on average. By controlling other factors of the current experimental paradigm, more precise comparisons between populations would be achievable, including comparison between populations with differing levels of expertise, different linguistic backgrounds, and different neurological profiles.

Lastly, the AOIs we constructed for these items were designed to capture what we conjectured to be minimal "chunks" of processing that could be likened to words in natural texts; however, more work is needed to truly understand if programming languages have such an equivalent. For example, we treat the chunk "range(len(word))" as a single AOI, but it may be of interest to researchers to delineate this even further such that "range," "len", and "word" are all separate AOIs. Indeed, at some level each word in this chunk likely requires lexical access similar to language processing (e.g., Duffy et al., 1988), but we decided to analyze this as one coherent chunk since it is quite commonly found in Python source code programming.

Moreover, even in the reading of natural texts, words are not always the base unit of processing since they themselves are made up of morphemes, which are subsequently made up of letters. Therefore, a future line of work could establish whether source code reading indeed uses the same cognitive categorization of linguistic chunks as is theorized to be used in natural reading.

Conclusion

Reading Python source code for debugging is a distinct cognitive process that utilizes many of the mechanisms underlying natural language text reading while also relying on knowledge of a separate, human–computer interaction system to transfer human instructions to something a computer can execute. The global reading patterns reported here can inform future research and updates in improving troubleshooting strategies and educational interventions while simultaneously providing a baseline for future source code reading research. More than anything, the current work presents the case for continued research investigating the real-time processes at work while programmers read source code, both for troubleshooting and for comprehension.

ACKNOWLEDGMENTS

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

REFERENCES

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*(6), 543-554.

Bürkner, P. C. (2017). brms: An R package for Bayesian multilevel models using Stan. *Journal of Statistical Software, 80*, 1-28.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... & Tamm, S. (2015, May). Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension* (pp. 255-265). IEEE.

Casalnuovo, C., Barr, E. T., Dash, S. K., Devanbu, P., & Morgan, E. (2020a, October). A theory of dual channel constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (pp. 25-28). IEEE.

Casalnuovo, C., Lee, K., Wang, H., Devanbu, P., & Morgan, E. (2020b). Do programmers prefer predictable expressions in code?. *Cognitive Science, 44*(12), e12921.

Casalnuovo, C., Morgan, E., & Devanbu, P. (2020c). Does surprisal predict code comprehension difficulty. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society.* Toronto, Canada: Cognitive Science Society.

Casalnuovo, C., Sagae, K., & Devanbu, P. (2019). Studying the difference between natural and programming language corpora. *Empirical Software Engineering, 24*(4), 1823-1868.

Christianson, K. (2016). When language comprehension goes wrong for the right reasons: Good-enough, underspecified, or shallow language processing. *Quarterly Journal of Experimental Psychology, 69*(5), 817-828.

Christianson, K., Dempsey, J., Tsiola, A., & Goldshtein, M. (2022). What if they're just not that into you (or your experiment)? On motivation and psycholinguistics. In *Psychology of Learning and Motivation-Advances in Research and Theory*. Academic Press Inc.

Christianson, K., Dempsey, J., M. Deshaies, S. E., Tsiola, A., & Valderrama, L. P. (2023). Do readers misassign thematic roles? Evidence from a trailing boundary-change paradigm. *Language, Cognition and Neuroscience, 38*(6), 872-892.

Christianson, K., Dempsey, J., Tsiola, A., Deshaies, S. E. M., & Kim, N. (2024). Retracing the garden-path: Nonselective rereading and no reanalysis. *Journal of Memory and Language, 137*, 104515.

Clifton Jr, C., Ferreira, F., Henderson, J. M., Inhoff, A. W., Liversedge, S. P., Reichle, E. D., & Schotter, E. R. (2016). Eye movements in reading and information processing: Keith Rayner's 40 year legacy. *Journal of Memory and Language, 86*, 1-19.

Crosby, M. E., Scholtz, J., & Wiedenbeck, S. (2002, June). The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *PPIG* (p. 5).

Dempsey, J., & Brehm, L. (2020). Can propositional biases modulate syntactic repair processes? Insights from preceding comprehension questions. *Journal of Cognitive Psychology, 32*(5-6), 543-552.

Drewnowski, A., & Healy, A. F. (1980). Missing-ing in reading: Letter detection errors on word endings. *Journal of Verbal Learning and Verbal Behavior, 19*(3), 247-262.

Duffy, S. A., Morris, R. K., & Rayner, K. (1988). Lexical ambiguity and fixation times in reading. *Journal of Memory and Language, 27*(4), 429-446.

Federmeier, K. D. (2007). Thinking ahead: The role and roots of prediction in language comprehension. *Psychophysiology, 44*(4), 491-505.

Fedorenko, E., Ivanova, A., Dhamala, R., & Bers, M. U. (2019). The language of programming: a cognitive perspective. *Trends in Cognitive Sciences, 23*(7), 525-528.

Ferreira, F., Bailey, K. G., & Ferraro, V. (2002). Good-enough representations in language comprehension. *Current Directions in Psychological Science, 11*(1), 11-15.

Francis, W. N., & Kucera, H. (1979). Brown corpus manual. *Letters to the Editor, 5*(2), 7.

Frazier, L., & Rayner, K. (1982). Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology, 14*(2), 178-210.

Frazier, L., & Rayner, K. (1987). Resolution of syntactic category ambiguities: Eye movements in parsing lexically ambiguous sentences. *Journal of Memory and Language, 26*(5), 505-526.

Gibson, E., Piantadosi, S. T., Brink, K., Bergen, L., Lim, E., & Saxe, R. (2013). A noisy-channel account of crosslinguistic word-order variation. *Psychological Science, 24*(7), 1079-1088.

GitLab (2021). Mapping the DevSecOps Landscape. https://about.gitlab.com/developer-survey/previous/2020/.

Herman, G. L., Meyers, S., & Deshaies, S. E. (2021, July). A Comparison of Novice Coders' Approaches to Reading Code: An Eye-tracking Study. In *2021 ASEE Virtual Annual Conference Content Access.*

Inhoff, A. W., & Rayner, K. (1986). Parafoveal word processing during eye fixations in reading: Effects of word frequency. *Perception & Psychophysics, 40*(6), 431-439.

Jacob, G., & Felser, C. (2016). Reanalysis and semantic persistence in native and non-native garden-path recovery. *Quarterly Journal of Experimental Psychology, 69*(5), 907-925.

Jbara, A., & Feitelson, D. G. (2017). How programmers read regular code: a controlled experiment using eye tracking. *Empirical Software Engineering, 22*(3), 1440-1477.

Kaakinen, J. K., & Hyönä, J. (2010). Task effects on eye movements during reading. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 36*(6), 1561.

Knuth, D. E. (1984). Literate programming. *The computer Journal, 27*(2), 97-111.

Kuperberg, G. R., & Jaeger, T. F. (2016). What do we mean by prediction in language comprehension?. *Language, Cognition and Neuroscience, 31*(1), 32-59.

Kutas, M., & Hillyard, S. A. (1984). Brain potentials during reading reflect word expectancy and semantic association. *Nature, 307*(5947), 161-163.

Levy, R. (2008, October). A noisy-channel model of human sentence comprehension under uncertain input. In *Proceedings of the 2008 conference on empirical methods in natural language processing* (pp. 234-243).

Levy, R., Bicknell, K., Slattery, T., & Rayner, K. (2009). Eye movement evidence that readers maintain and act on uncertainty about past linguistic input. *Proceedings of the National Academy of Sciences, 106*(50), 21086-21090.

Lim, J. H., & Christianson, K. (2015). Second language sensitivity to agreement errors: Evidence from eye movements during comprehension and translation. *Applied Psycholinguistics, 36*(6), 1283-1315.

Lovric, M. M. (2020). Conflicts in Bayesian statistics between inference based on credible intervals and Bayes factors. *Journal of Modern Applied Statistical Methods, 18*(1), 14.

Luke, S.G. & Christianson, K. (2016). Limits on lexical prediction during reading. *Cognitive Psychology, 88,* 22-60.

Osterhout, L., & Holcomb, P. J. (1992). Event-related brain potentials elicited by syntactic anomaly. *Journal of Memory and Language, 31*(6), 785-806.

Peitek, N., Siegmund, J., & Apel, S. (2020, July). What drives the reading order of programmers? An eye tracking study. In *Proceedings of the 28th International Conference on Program Comprehension* (pp. 342-353).

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295-341.

Pickering, M. J., & Traxler, M. J. (1998). Plausibility and recovery from garden paths: An eye-tracking study. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 24*(4), 940.

R Core Team (2021). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin, 124*(3), 372.

Rayner, K., Chace, K. H., Slattery, T. J., & Ashby, J. (2006). Eye movements as reflections of comprehension processes in reading. *Scientific Studies of Reading, 10*(3), 241-255.

Rayner, K., & Duffy, S. A. (1986). Lexical complexity and fixation times in reading: Effects of word frequency, verb complexity, and lexical ambiguity. *Memory & Cognition, 14*(3), 191-201.

Rayner, K., Pollatsek, A., Ashby, J. & Clifton, C. E., Jr. (2012). *The psychology of reading.* New York: Psychology Press.

Reichle, E. D., Rayner, K., & Pollatsek, A. (2003). The EZ Reader model of eye-movement control in reading: Comparisons to other models. *Behavioral and Brain Sciences, 26*(4), 445-476.

Reichle, E. D., & Reingold, E. M. (2013). Neurophysiological constraints on the eye-mind link. *Frontiers in Human Neuroscience, 7*, 361.

Rayner, K., & Reingold, E. M. (2015). Evidence for direct cognitive control of fixation durations during reading. *Current Opinion in Behavioral Sciences, 1*, 107-112.

Schotter, E. R., Angele, B., & Rayner, K. (2012). Parafoveal processing in reading. *Attention, Perception, & Psychophysics, 74*, 5-35.

Schotter, E. R., Bicknell, K., Howard, I., Levy, R., & Rayner, K. (2014). Task effects reveal cognitive flexibility responding to frequency and predictability: Evidence from eye movements in reading and proofreading. *Cognition, 131*(1), 1-27.

Shaft, T. M., & Vessey, I. (1995). The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research, 6*(3), 286-299.

Sharif, B., Falcone, M., & Maletic, J. I. (2012, March). An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (pp. 381-384).

Schneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences, 8*(3), 219-238.

Slattery, T. J., Sturt, P., Christianson, K., Yoshida, M., & Ferreira, F. (2013). Lingering misinterpretations of garden path sentences arise from competing syntactic representations. *Journal of Memory and Language, 69*(2), 104-120.

Slattery, T. J., & Yates, M. (2018). Word skipping: Effects of word length, predictability, spelling and reading skill. *Quarterly Journal of Experimental Psychology, 71*(1), 250-259.

Stites, M. C., Federmeier, K. D., & Stine-Morrow, E. A. (2013). Cross-age comparisons reveal multiple strategies for lexical ambiguity resolution during natural reading. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 39*(6), 1823.

Strukelj, A., & Niehorster, D. C. (2018). One page of text: Eye movements during regular and thorough reading, skimming, and spell checking. *Journal of Eye Movement Research, 11*(1).

Turner, R., Falcone, M., Sharif, B., & Lazar, A. (2014, March). An eye-tracking study assessing the comprehension of C++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (pp. 231-234).

Von der Malsburg, T. (2022). Python script that uses NLTK to calculate the average length of English words across token and types in the Brown corpus. https://gist.github.com/tmalsburg/366e167f80f76dca5ea2e68d858ee845

Vee, A. (2017). *Coding literacy: How computer programming is changing writing*. MIT Press.