Noname manuscript No.
(will be inserted by the editor)

An Exploratory Eye Tracking Study on How Developers Classify and Debug Python Code in Different Paradigms

Samuel W. Flint · Jigyasa Chauhan · Niloofar Mansoor · Bonita Sharif · Robert Dyer

Received: date / Accepted: date

Abstract Modern programming languages, such as Python, support language features from several paradigms, such as object-oriented, procedural, and functional. Research has shown that code written in some paradigms can be harder to comprehend, but to date, no research has looked atwhich paradigm-specific language features impact comprehension. To this end, this study seeks to uncover which paradigm-specific features impact comprehension and debugging of code or how multi-paradigm code might affect a developer's ability to do so. We present an exploratory empirical eye-tracking study to investigate 1) how developers classify the predominant paradigm in Python code and 2) how the paradigm affects their ability to debug Python code. The goal is to uncover if specific language features are looked at more often while classifying and debugging code with a predominant paradigm. Twenty-nine developers (primarily students) were recruited for the study and were each given four classification and four debugging tasks in Python. Eye movements were recorded during all the tasks. The results indicate confusion in labeling Functional and Procedural paradigms, but not Object-Oriented. The code with predominantly functional

The majority of this work was completed while S. Flint was at the University of Nebraska–Lincoln.

Corresponding author: Samuel W. Flint
Dakota State University
E-mail: Samuel.Flint@dsu.edu

Co-first-author: Jigyasa Chauhan
University of Nebraska–Lincoln
E-mail: chauhan@huskers.unl.edu

Niloofar Mansoor
University of Nebraska–Lincoln
E-mail: niloofar@huskers.unl.edu

Bonita Sharif
University of Nebraska–Lincoln
E-mail: bsharif@unl.edu

Robert Dyer
University of Nebraska–Lincoln
E-mail: rdyer@unl.edu

paradigms also took the longest to complete. Changing the predominant paradigm did not affect the ability to debug the code, though developers did rate themselves with lower confidence for Functional code. We report significant differences in reading patterns during debugging, especially in the Functional code. During classification, results show that developers do not necessarily read paradigm-relevant token types.

1 Introduction

Most modern programming languages have a predominant programming paradigm they support, often object-oriented. Yet they are also multi-paradigm and support other programming paradigms, such as procedural or functional, in addition to its predominant paradigm. Python, a popular and currently one of the fastest growing programming languages (TIOBE Software BV, 2023; Carbonnelle, 2023; Github, 2022), is a good example of such a multi-paradigm language. To date, little is known about how developers utilize the different programming paradigms in multi-paradigm languages, such as Python.

Determining which paradigm some code is a part of helps developers select the right data structure(s) and algorithm(s) for the problem at hand. Thus, code classification is an important part of how developers comprehend code (Floyd, 1979). Knowing the paradigm used also explains how developers build mental models of the code (Brooks, 1983; Letovsky, 1987; Rist, 1986; Soloway and Ehrlich, 1984; Pennington, 1987; Von Mayrhauser and Vans, 1995). Even though this is not something a programmer consciously performs in practice as a concrete, high-level task, it is a precursor to program understanding and other tasks such as bug fixing, adding features, refactoring, etc. It is implicit and embedded into the developer workflow (Wells and Kurtz, 1989).

An initial study in this direction (Dyer and Chauhan, 2022) investigated how 100K open-source Python projects utilize procedural, object-oriented, functional, and imperative paradigms in their source files and at the project level. They manually classified a small sample of source files and developed an automated classification script that runs in the Boa language (Dyer et al., 2013; Rajan et al., 2021). During the manual process, they noted that human raters did not fully agree on what the predominant paradigm was and it was not clear how humans were even judging the predominant paradigm in Python code. They also did not go beyond simply classifying the paradigm(s) observed. It is still not understood if the predominant paradigm of Python scripts hinders a developer's comprehension or their ability to debug that code.

Python provides a variety of language features with respect to each paradigm. For example, object-oriented features such as classes, functional features such as lambdas, procedural features such as function definitions, and imperative features such as names and assignments. Understanding how developers use the different programming paradigm features available to them and the possible effects of those paradigms on their comprehension is an important topic to study. It has been observed that the first language taught to students in early courses of computer science affects their comprehension of programming (Siegfried et al., 2019) and thus developing Python code in a predominant paradigm that is different than their first language could have an effect on learning. When a developer transitions from an imperative paradigm to

a more advanced one like object-oriented or functional, it becomes harder for them to grasp new concepts associated with those features (Floyd, 1979).

Further studies demand the need to understand Python features (Shrestha et al., 2020) among the developer community, for example, researchers exploring pattern matching using functions, first-class objects along with pre-existing features (Kohn et al., 2020), dynamic analysis frameworks supporting Python features like lambda (Eghbali and Pradel, 2022), or studies that are using Python packages for analyzing relationship structure using feature definitions, imports, and calls (Romanov, 2020). Such knowledge can also help guide future researchers investigating code smells in Python (Chen et al., 2016; Rahman et al., 2019) as they could correlate existing smells to specific paradigms or see if new paradigm-specific smells are occurring. These problems inspired us to look into how developers understand Python's paradigm-specific language features.

The study we present in this paper explores the question: "How do developers understand Python's paradigm-specific language features?" In particular, we explore the comprehension of paradigm-specific language features through two programming tasks: classification of paradigm and bug localization. This helps us better understand how developers read code (Just and Carpenter, 1980), using an eye tracker (Duchowski, 2017), that collects time-stamped events on code tokens. This methodology has been used within the software engineering community over several years to understand developers' thought processes (Obaidellah et al., 2018; Sharafi et al., 2015a). Eye tracking has provided valuable insights into how developers work and how their thought processes differ based on factors such as expertise or task (Abid et al., 2019) all within a realistic development environment. In particular, the data from this study will help us build new and refine older theories of program comprehension (Storey, 2005) for mixed paradigm languages, which are sorely lacking within the field, and may help us to design more usable programming languages, as well as improve education. This study was performed in a realistic development environment (an IDE), with 29 participants of different experience levels in Python. All participants were trained to classify Python features by paradigm and asked to complete the two programming tasks. We find the following key results:

1. During classification, developers appear unsure between Functional and Procedural paradigms but can recognize Object-Oriented code.
2. During classification, the length of the code appears to drive differences in reading behavior more than the predominant paradigm, but the fixation counts and durations are not linearly correlated with the code length.
3. During classification, developers fixate on many different token types in all paradigms, but they do not only focus on paradigm-relevant token types.
4. During debugging, the paradigm of the code influences how accurately developers can spot the bug (Procedural code in particular shows the highest accuracy), however, the particular program being debugged has an impact as well.
5. During debugging, we find higher fixation counts, longer fixation durations, and significantly different vertical and horizontal reading patterns for Functional code.

In the next section, we provide background on the classification of predominant paradigms in Python code. In Section 3 we discuss some related prior studies. The studied research questions are provided in Section 4. The study design is given in Section 5 followed by the results in Section 6. Section 7 provides a discussion of the

Listing 1: Example Python code with each statement classified into paradigm(s) (based on Dyer and Chauhan (2022))

```
1  class MyCounter:                                  # func oo
2     x = 1                                          #      oo      imp
3     def val(self):                                 #      oo
4        def wrapper():                              #      oo proc
5           return self.x                            #      oo proc
6        return wrapper                              # func oo
7     def __iter__(self):                            # func oo
8        self.x = 1                                  #      oo
9        return self                                 #      oo
10    def __next__(self):                            # func oo
11       if self.x > 50:                             #      oo
12          raise StopIteration                      #      oo
13       tmp = self.x                                #      oo
14       self.x += 1                                 #      oo
15       return tmp                                  #      oo
16 print([y for y in MyCounter() if y % 2 == 0]) # func oo proc
```

results with threats to the validity discussed in Section 8. Conclusions and future work are presented in Section 9.

2 Background on Paradigm Classification

Python is predominantly object-oriented but also supports other paradigms like procedural and functional. For example, the Django framework (Foundation, 2021) supports class-based and function-based views. Some articles claim function-based views are easier to read and understand (Freitas, 2021), but the framework provides both and is moving more toward the class-based paradigm. Here we briefly describe some of the features Python supports and how we can classify Python code according to those features based largely on the work of Dyer and Chauhan (2022). We re-use and slightly modify the running example from their work and show it in Listing 1 and explain how they would classify this code. Their classification strategy is based on specific features, where they mapped features as functional based on the Python official documentation's how-to guide for functional programming (Kuchling, 2021). The full feature mapping is shown in Dyer and Chauhan (2022, Tab. 1).

We can classify the overall code in Listing 1 as object-oriented since the majority of the code implements a class and uses the class (on line 16). Thus every line has an `oo` classification associated with it. Procedural programming supports defining functions which are known as procedures. When we define a function `def():` we can pass different arguments. On lines 4 and 5, we classify them as procedural. There is also a procedural call to the print function on line 16. Since iteration is considered functional, we also classify the class itself and the methods providing iteration support (lines 1, 7, 10) as functional. But the `wrapper` function is being returned in the method, on line 6. This is an example of a higher-order function and thus we classify line 6 as also being functional.

We can already see that several lines are classified as belonging to multiple paradigms. Most lines in Python are also imperative by nature, either by referencing names or using assignments. As such, we could potentially mark almost every line as imperative. Instead, the prior approach states that things should only be marked as imperative if they were not already classified as another paradigm. Thus, line 2

is also marked as imperative because the object-oriented classification of line 2 only came about from the block structure of the code and not from the line in isolation. In general, very few lines wind up being classified as imperative and thus very few files are predominantly imperative, using this prior classification strategy (Dyer and Chauhan, 2022). In this study, we ignore the imperative features and focus only on procedural, object-oriented, and functional features. Finally, the last line is also functional as it contains a list comprehension.

Since each line can be classified into more than one paradigm, it is possible for several paradigms to account for more than half the code. We follow the classification strategy of Dyer and Chauhan (2022) to indicate when a file is *mixed paradigm*: if the top two paradigms are each present on 2/3 of the lines, or present on 1/2 the lines and within 20% of each other, or if the most-used paradigm accounts for less than half the lines and the second most-used is within 10%. For all other cases, we classify the code by its *predominant paradigm* (the most-occurring paradigm).

The Boa script from Dyer and Chauhan (2022) is also used in our study to classify each file. Their script relies on the classifications from Listing 1 as well as some heuristics they discovered during their manual classification process (such as the fact that we do classify line 2 as imperative, but most other lines are not). This script is used to classify every source file used in our study and forms our ground truth for the classification and debugging tasks.

3 Related Work

We first describe prior work related to Python language feature studies followed by relevant eye-tracking studies done in software engineering.

3.1 Language Feature Studies in Python

Hansen et al. (2013) conducted an experiment on 10 Python programs that varied in notation to determine which variation was harder to understand. They found that experienced participants performed better but not when certain expectations were violated in the code. They did not consider paradigm classification or feature use in their tasks. Next, Peng et al. (2021) investigated using several different Python features, such as loops, nested classes, inheritance, and keyword arguments. They built a tool to automatically identify and measure the use of these features in 35 popular projects. While the features they studied crossed paradigms, they did not focus on how paradigm-specific subsets were used or the impact of those paradigm-specific features on developers.

Similarly, Yang et al. (2022) looked at "complex" language features, such as reflection, dynamic features, context managers, and generators. Although the set of features they studied overlaps with several paradigms (object-oriented and functional), their focus was not on the impact of paradigms, but the impact of features. This work did not perform a user study to determine how those features impact developers, while our study aims to see if paradigms affect developer comprehension of code. Further studies, such as performed by Keshk and Dyer (2023) studied one specific feature in three languages, including Python: the ability to chain method calls together. They showed the use of method chains differs in Python compared

to other languages like Java. Their study used repository mining techniques and did not investigate what impact the use of chaining might have on developers.

Finally, there has also much research on idiomatic code in Python (Alexandru et al., 2018; Zhang et al., 2022, 2023a,b; Farooq and Zaytsev, 2021). While some research has looked at identifying idioms in Python, other work has attempted to transform existing code to be more idiomatic. All of these works looked at idioms instead of specific language features, and most only statically analyzed existing repositories instead of performing a user study to measure the potential impact on developers. The study presented in this paper bridges this gap in prior literature by providing objective metrics on what Python developers actually look at while solving tasks. It follows a developer-centric instead of an artifact-centric approach.

3.2 Eye-tracking Studies

Eye tracking has been used for a number of years to explore text comprehension, with wide-ranging results (Rayner, 1998). Additionally, it has been used for a number of software engineering studies, many of which have been conducted using the Java language (Sharafi et al., 2015b; Obaidellah et al., 2018). In particular, studies have been conducted on various topics such as identifier style (Binkley et al., 2013), reading order (Peitek et al., 2020; Busjahn et al., 2015), bug fixing (Kevic et al., 2015), scope highlighting (Park et al., 2023), code summarization (Rodeghero et al., 2014; Abid et al., 2019), atoms of confusion (de Oliveira et al., 2020; da Costa et al., 2023), and the very first paper on eye tracking in comprehension by Crosby and Stelovsky (1990).

We are aware of two eye-tracking studies done on Python and present them next. Turner et al. conducted a study on small code snippets in Python Turner et al. (2014) that contained bugs. They found no significant difference in bug-finding accuracy or time, however, there was a difference in the rate at which buggy lines were looked at—one difference being that novices had higher fixation durations on buggy lines in Python. In recent work, da Costa et al. conducted a controlled experiment with 32 novices in Python using an eye tracker (da Costa et al., 2023) and found that code that had operator precedence took more attempts to solve. In contrast, code with multiple variable assignments had 60% fewer regressions, indicating more clarity. We are unaware of other eye-tracking studies focusing on feature classification and debugging in Python. Our study seeks to bridge this gap by studying the eye movements of developers as they classify and debug in Python. We significantly add more depth to prior analyses by using additional linearity metrics (Busjahn et al., 2015) and scanpath visualizations via scarf plots (Yang and Wacharamanotham, 2018).

3.3 Bugginess and Language Paradigms

Prior research has explored the effect of paradigm (and other language features) on the presence of bugs, with some research finding an increase in bugs in the presence of functional code (Ray et al., 2014). In contrast, later research found fewer associations with bugginess (Berger et al., 2019). This work in contrast looks at a single language

and examines how effectively developers can detect bugs in code written in different paradigms within the same language.

4 Research Questions

As the main goal of this study is to uncover how paradigm-specific language features impact developer comprehension, we pick two important programming tasks to study. The first is paradigm classification, which plays an early role in developer comprehension; here we seek to understand how well developers classify paradigms, and how their reading behavior changes between different paradigms. We devise the following two research questions for paradigm classification.

RQ1 How accurately and efficiently do Python developers classify the predominant paradigm? To explore this, we use a questionnaire and timing information, to understand ability and efficiency.

RQ2 How do different predominant paradigms impact developers' reading when classifying for predominant paradigm? For example, do developers look at paradigm-specific tokens? Are developers' reading patterns more or less linear in certain paradigms? To explore this, we use eye tracking data, including fixations-on-AOIs (areas of interest), and various linearity metrics.

With these results in mind, we want to understand how the paradigm of code affects a particular downstream task: localization of logical errors in code. This is an important downstream task, and we explore it both using the lenses of efficiency and gaze studied via the following two research questions:

RQ3 How do different predominant paradigms affect a Python developer's ability and efficiency to debug logical errors in code? To explore this, we use a questionnaire and timing information, to understand ability and efficiency.

RQ4 How do different predominant paradigms impact developers' reading when debugging logical errors? For example, do developers look at paradigm-specific tokens? Are developers' reading patterns more or less linear in certain paradigms? To explore this, we use eye tracking data, including fixations-on-AOIs (areas of interest), and various linearity metrics.

In this exploratory study, we hope to start gaining some insight into how code written predominantly in one of the paradigms namely, Python, supports influences the comprehension of that code (RQ1 and RQ3). Since Python is a multi-paradigm language, it is unclear if developers know what paradigm is being predominantly presented or if the token-level language features help them determine that. It is also not clear if writing code in one specific paradigm over another leads to lower comprehension of errors occurring in that code. With RQ2 and RQ4, we also seek to uncover via eye tracking how developers actually read and navigate the code in the different paradigms for classification and debugging tasks and use specific metrics to operationalize the reading behavior.

5 Study Design

This section discusses our study design, tasks, participants, procedures, and analysis methods. All de-identified study data and processing scripts are available in our replication package (Flint et al., 2023).

5.1 Terminology

We define the following terms for use in this paper.

logical bug A non-syntactic bug, violating the semantics or logic of the task, for example, computing $x \times x \times x \times 3$ to cube a value instead of computing $x \times x \times x$ or x^3 (see Fig. 1).

code paradigm One of "functional", "object-oriented", "procedural", or "imperative". This taxonomy, and its definition, was developed previously by Dyer and Chauhan (2022).

5.2 Task Categories

We divided study tasks into two categories: classification and bug localization. Each task category had four code snippets with corresponding questions for each. Each code snippet was shown as several variants. The variant for the classification task was the length of the code snippet in a specific paradigm, whereas the variant for the bug localization task was the program shown in a specific paradigm. Each participant was first shown four questions from the classification task category in the functional, mixed, object-oriented (OO), and procedural paradigms with a randomly chosen code length (small, medium, large). The second set of tasks was from the bug localization category, where the programs (cube, largest, palindrome, factorial) were randomly selected in the various paradigms. Refer to Table 1 for a sequence of how the questions in each task category were assigned to participants. We describe each of the task categories next.

Classification Task Category. The first task category was designed to support answering RQ1–RQ2. These questions are classification tasks, where participants are asked to classify the predominant paradigm (Functional, Procedural, or Object-Oriented) in Python code. We obtained real-world Python source file URLs using Boa's "2021 Aug/Python" dataset (Dyer et al., 2013). From that dataset, files were chosen in three different length categories. The lengths were determined using the number of statements in the file (as Boa does not provide line numbers), with a goal of having lengths that are less than a screen of code, about a full screen, and more that would require scrolling. There were three lengths: *Small* (10-15 statements), *Medium* (16-30 statements), and *Large* (31-45 statements). There were a total of four code snippets (Procedural, Object-Oriented, Functional, and Mixed) and each of them was accompanied by a set of survey questions. Participants were asked to classify the code examples based on the predominant paradigm and choose what percent of the code belonged to each paradigm. An example of the classification task can be seen in Listing 1, where each statement is classified into a programming paradigm(s).

Table 1: Assigning participants to specific tasks. The bold lines show the sequence for Participant 2.

	1. Classification Tasks			
Participant	**A. Functional**	**B. Mixed**	**C. OO**	**D. Procedural**
1	Small	Medium	Large	Small
2	**Medium**	**Large**	**Small**	**Medium**
3	Large	Small	Medium	Large
⋮	⋮	⋮	⋮	⋮
29	Medium	Large	Small	Medium
	2. Bug Localization Tasks			
	A. Cube	**B. Factorial**	**C. Largest**	**D. Palindrome**
1	Functional	OO	Procedural	Mixed
2	**OO**	**Procedural**	**Mixed**	**Functional**
3	Procedural	Mixed	Functional	OO
⋮	⋮	⋮	⋮	⋮
29	OO	Procedural	Mixed	Functional

Bug Localization Task Category. The second task category was designed to support answering RQ3–RQ4. These tasks are bug localization tasks to measure how well participants comprehend the given code in a specific paradigm. Participants were asked to find and explain a bug in the given source code. The code's first two lines stated the purpose of the code. There were four different variants of programs provided: *Cube* calculates the cube of a number, *Largest* finds the largest number in a list, *Palindrome* checks if the input string is a palindrome, and *Factorial* finds the factorial of a number. For each, a single logical bug (a non-syntactic bug, which violates the semantics or logic of the task) was introduced. We had four variants of each program, one for each paradigm. Each variant had the same logical bug applied to it, expected the same input, and generated the same output. An example of a bug localization task is shown in Figure 1, where the bug was the addition of "`* 3`" when cubing the value.

5.3 Study Variables

Independent Variables. The common independent variable for the classification and bug localization tasks is the *paradigm* the code was shown in. The paradigm's treatment in both task categories was one of four types: Functional, Mixed, Object-Oriented, and Procedural. In addition, the classification task had another independent variable: *code length*—with Small, Medium, and Large treatments. The bug localization task exchanged code length for a different independent variable: *program*—with Cube, Factorial, Largest Number, and Palindrome treatments. The study used a within-subjects design for each independent variable (paradigm, code length, and program). This design allows us to better explore variation caused by the different factors, mitigate learning effects, and improve sampling efficiency (allowing us to maximally utilize limited participant time).

```
items = [1, 2, 3, 4]

_ = list(map(print, [x * x * x * 3 for x in
      items]))
```

(a) Functional code

```
items = [1, 2, 3, 4]
for i in items:
   print(i * i * i * 3)
```

(b) Mixed-paradigm code

```
items = [1, 2, 3, 4]
class Cuber:
   def __init__(self, x):
      self.x = x

   def cube(self):
      print(self.x * self.x * self.x * 3)

for number in items:
   c = Cuber(number)
   c.cube()
```

(c) Object-Oriented code

```
items = [1, 2, 3, 4]
def cube(x):
   return x * x * x * 3

for number in items:
   res = cube(number)
   print(res)
```

(d) Procedural code

Fig. 1: The *Cube* program for the bug localization task category, shown in all four paradigms. Note that each file had a two-line header describing the task ("Find a logical bug in the code"), and the purpose of the code ("The following code prints the cubed values of a list of numbers").

Dependent Variables. We analyze several dependent variables (detailed in Table 2) namely correctness, time, confidence, and eye tracking metrics including scan paths (Sharafi et al., 2015a). The answers to the questionnaire were used to collect correctness, time, and confidence (measured on a five-point Likert-type scale from "Not confident" to "Very confident"). The use of time is a proxy for task difficulty, and developer performance: faster completion likely indicates that code was easier to understand. The raw gaze data was used to generate fixations and saccades (Andersson et al., 2017). We measured fixation-based and linearity metrics (Busjahn et al., 2015) such as the amount of time developers spent reading a line, moving up/down in the code, re-reading code, and navigation (denoted by saccade length) using the eye-tracking data. Fixations indicate focus and attention, and fixation-based metrics (Andersson et al., 2017) are widely used in eye-tracking studies to analyze code reading behavior and attention (Sharif and Maletic, 2010; Busjahn et al., 2014, 2015; Abid et al., 2019; Beelders and du Plessis, 2016). Such metrics are associated with reading behaviors which are influenced by program design, and may be influenced by paradigm-specific tokens or constructs. This paper uses fixation count, fixation duration, and normalized mean fixation duration. For normalization, each fixation duration is divided by the number of characters of the fixated token, then the mean is taken. This allows us to consider the amount of time spent on paradigm-specific tokens while avoiding issues caused by token length. In addition to the fixation metrics, we use the linearity metrics designed by Busjahn et al. (2015). The linearity metrics show vertical and horizontal reading patterns, saccade length (length between two consecutive fixations), and regressions. We used the variables collected from the questionnaire to answer RQ1 and RQ3, and the eye-tracking data variables to answer RQ2 and RQ4. These reading patterns should vary by paradigm, as differ-

ent paradigms follow more or less linear patterns of organization, or exhibit different patterns of information locality.

Table 2: Dependent variables and their measures, mapped to research questions.

RQs	Variable	Measure
1, 3	correctness of response	choice of paradigm
1, 3	confidence of response	5 point Likert-type
1, 3	time to completion	time in seconds
2, 4	fixation count	count (frequency)
2, 4	fixation duration	time in milliseconds (ms)
2, 4	vertical next text	forward saccades (%) on same or one line down
2, 4	vertical later text	forward saccades (%) on same or any number of lines down
2, 4	horizontal later text	forward saccades (%) on same line
2, 4	line regression rate	backward saccades (%) on same line
2, 4	saccade length	mean Euclidean distance between consecutive fixations
2, 4	scanpaths	scarf plots

5.4 Participants and Grouping

We recruited participants 19 years of age or older from the University of Nebraska–Lincoln, requiring self-reported prior Python experience. This gave us a total of 31 participants, but two had to be excluded: one for poor calibration, and one for having no prior Python experience. We analyzed the data from the remaining 29 participants. The majority of participants were students from the University of Nebraska–Lincoln. The majority of the students were from a computer science background and others were from biological systems engineering, statistics, computer engineering, and other related engineering fields. Because of varying backgrounds, we cannot verify programming language background or history of the participants. The participants were not provided any course incentives but were compensated with a $25 virtual Visa card.

Each participant was provided with twelve questions in total, six in each task category. Participants provided their responses to each question by typing their answers into a web form using a provided tablet. Classification tasks were assigned to ensure that roughly one-third of participants saw a small+functional code, one-third saw medium+functional, and one-third saw large+functional. This was done for each of the four paradigms. Similarly, for the bug localization category, tasks were assigned to ensure roughly equal coverage among each program and paradigm. Table 1 shows an example grouping for four participants.

5.5 Eye-Tracking Apparatus and Environment

To track the participants' eyes, we used the Tobii TX300 eye tracker running at 60Hz frequency. Participants used Eclipse as the integrated development environment (IDE) for viewing all code snippets. The iTrace infrastructure (Guarnera et al., 2018; Zyrianov et al., 2020) was used to map the raw eye coordinates to a line and

column in the file as the participant looked at the source code while performing tasks. The iTrace-Eclipse plugin provides an output of two XML files containing time-stamped session logs. The data is then passed through the Identification by Dispersion Threshold (IDT) fixation filter (Andersson et al., 2017) with the dispersion threshold parameter set to 125 pixels and the duration threshold set to 10 milliseconds using iTrace-Toolkit (Behler et al., 2023) to generate the fixations over source code. At the time of this writing, iTrace does not natively support mapping of eye gaze to source tokens for Python. We wrote a Python script to generate the mapping from gazes to tokens using the line and column information generated in the iTrace databases. Other than mapping tokens, we performed no other post-processing of eye-tracking data.

5.6 Study Procedure

The study was conducted in a dedicated eye-tracking lab. Participants were allocated up to two hours to complete the entire study. After signing a consent form, participants filled out the pre-questionnaire at the start with some demographic information. They were then given the training task. Next, each participant had to be calibrated with the eye tracker, where they learned how to sit and position themselves in front of the monitor properly. Once the calibration was done, the participant was ready to perform tasks. For each task, the participants were asked to read code in Eclipse and answer questions about the code. Each task was allocated a maximum of 10 minutes for participants to complete. After each task category, a debriefing was done as part of the protocol. Finally, participants filled out a post-questionnaire where they self-rated their Python expertise levels. We did this last to avoid any stereotype threats (Spencer et al., 1999) that might affect study performance.

5.6.1 Participant Training

In order to perform the study tasks, we first trained all participants. This process was started by first showing the participants a reference sheet that consisted of mock code snippets for the classification and debugging tasks. These mock snippets were built from Dyer and Chauhan (2022, fig. 1), which explained how the paradigms use Python features, by classifying each statement. They were also provided with the feature classification table as shown in Dyer and Chauhan (2022, Tab. 1), which was available at any time during the classification portion of the study. Here, during training, we introduced the concept of predominant paradigm, wherein they saw how the training code has a predominant paradigm of object-oriented since all statements are part of a `class MyNumbers` for instance. While understanding every feature-wise classification per statement, they were also made aware of how one statement can belong to multiple paradigms. There was also a bug localization training task where we seeded a fault by changing from `+=` to `-=` (see replication package Flint et al. (2023)). By training participants on how to classify features, they should pay more attention to paradigm-specific features, thus impacting downstream measures.

5.6.2 Analysis

To analyze the effects of the presented paradigm on correctness, we use Cochran's Q (Cochran, 1950), with individual subjects as blocks, and the presented paradigm

(or code length, or program) as treatments. To quantify the effect of treatment overall, we use the η^2_Q as proposed by Berry et al. (2007). To follow up on the omnibus Q, we use McNemar's χ^2 test (McNemar, 1947) and Bonferroni's correction to control the family-wise error rate. Finally, to analyze relationships between confidence and the presented paradigm, length, or program, we use Fisher's Exact Test (Fisher, 1922), because we have small cell counts (< 5) which would otherwise cause problems for the usage of Pearson's χ^2.

To analyze the effects of task and presented paradigm on time-on-task and various fixation and linearity metrics, we use within-groups factorial ANOVAs as omnibus tests, (Boslaugh, 2013, p. 210). For these analyses, we report degrees of freedom, the F statistic, and p values, as well as the effect sizes η^2, η^2_p (partial), and Cohen's f. We then follow-up the omnibus ANOVAs with pairwise comparisons performed using Tukey's HSD (Tukey, 1949), a single-step multiple comparisons test that produces adjusted p-values.

Finally, to determine if paradigm-specific tokens are fixated on longer, we use a simple t-test between count of fixations in and not in areas of interest (AOI) for each task/paradigm pair. We follow this up with Bonferonni's correction to avoid the family-wise error rate problem. A listing of paradigm-specific tokens, based on Dyer and Chauhan (2022) is shown in Table 3.

Table 3: Paradigm-Specific Tokens

Functional	**OO**	**Procedural**	**Mixed**
Lambda	Attribute	arg	Attribute
Call	ExceptHandler	Call	arg
GeneratorExp	Call	Return	Lambda
ListComp	Raise	FunctionDef	ExceptHandler
	With		Call
	FunctionDef		Return
	ClassDef		Raise
			With
			FunctionDef
			GeneratorExp
			ClassDef
			ListComp

Statistical analyses are performed using GNU R (R Core Team, 2018), with the `effectsize` package (Ben-Shachar et al., 2020) to calculate η^2, and the `nonpar` package (Sweet, 2020) to calculate Cochran's Q. Statistical visualizations are produced using `matplotlib` (Hunter, 2007) and `seaborn` (Waskom, 2021), with data handling provided by `pandas` (the pandas development team, 2020). Specific version information and environment definition are in our replication package (Flint et al., 2023).

In order to analyze scanpaths (how one reads i.e., moves from one fixation to next) for RQ2 and RQ4, we take advantage of the scarf plot visualization (Yang and Wacharamanotham, 2018). The scarf plot shows eye transitions between different parts of the code snippets and are mainly used to visualize gaze transitions within areas of interest in time. The x-axis in a scarf plot represents the task timeline showing duration of how long was spent viewing the token type in the code snippet. The y-axis shows each participant in the group on the timeline. The correctness

of each participant is also reported near the y-axis to allow to look for trends in eye movement scanpaths between correct vs. incorrect answers. See Figure 3 for an example of a scarf plot.

6 Results

6.1 Participant Demographics

To understand participant expertise in Python, we ask for a self-report in a post-questionnaire to avoid priming and stereotype effects (Steele and Aronson, 1995; Spencer et al., 1999; Shapiro and Neuberg, 2007). We observe that the majority rated themselves as having intermediate Python skills (69%), few had Expert-level skills (10%), with the remainder being Novice (21%). 58% of participants had two or less years of experience, while 31% had 2-5 years of experience. Only 10% of participants have five or more years of experience, which matches the number that self-reported as experts. These and other questions (such as how developers view coding in Python and how frequently they code in Python) are summarized in Table 4.

Table 4: Demographics for the 29 participants (after 2 excluded)

(a) "How would you rate your programming in Python?"

Novice	21%
Intermediate	69%
Expert	10%

(b) "How often do you code using Python?"

Very Frequently	17%
Frequently	34%
Occasionally	41%
Very Rarely	7%

(c) "How long have you been coding using Python?"

Less than or equal to 1 year	24%
Between 1-2 years	34%
Between 2-3 years	14%
Between 3-5 years	17%
Greater than or equal to 5 years	10%

(d) "How important is it for you to code in Python?"

Very Important	31%
Important	31%
Moderately Important	28%
Slightly Important	3%
Not Important	7%

6.2 RQ1 Results: Accuracy and Efficiency of Paradigm Classification

6.2.1 Classification Accuracy

We show the accuracy of judgment, broken down by both paradigm and code size, in Table 5. Of note, the presented paradigm does influence how effectively participants were able to classify code ($Q(2) = 9.50$, $p = .01$, $\eta^2 = 0.16$). In particular, we find that there is no difference between Object-Oriented code and Functional code in terms of the ability to classify ($\chi^2(1) = 1.13$, $p_{adj} = .87$), nor between Functional code and Procedural code ($\chi^2(1) = 2.08$, $p_{adj} = .45$), but there is a difference between Object-Oriented code and Procedural code ($\chi^2(1) = 6.75$, $p_{adj} = .03$). This

comparison shows that participants were able to correctly classify Object-Oriented code more often (86% of the time) than Procedural code (52% of the time).

Table 5: Accuracy of paradigm decision. Participants were asked: "What do you consider the predominant programming paradigm for the code you read?" The presented paradigm is shown in rows and code length is in columns. Highlighted cells represent stimuli categories that showed a statistical difference.

	Small	**Medium**	**Large**	**Overall**
Functional	8 / 10	5 / 10	8 / 9	72.41%
Object-Oriented	9 / 10	7 / 9	9 / 10	**86.21%**
Procedural	6 / 10	5 / 10	4 / 9	**51.72%**
Overall	79.31%	58.62%	72.41%	

To further explore this difference, we look at Table 6 which shows how participants binned code into paradigms. This shows us that Object-Oriented samples (row two) were consistently rated as being $> 75\%$ Object-Oriented, with participants consistently considering it as $\leq 50\%$ Functional or Procedural. Participants, however, were more confused by Functional (row one) and Procedural (row three) samples. Functional code was thought to be at least 75% Functional by most participants, but almost a third thought it was at least 75% Procedural; this pattern is flipped for Procedural.

Finding 1: When classifying the predominant paradigm, there appears to be confusion in labeling Functional and Procedural paradigms, while classifiers are accurate on Object-Oriented.

We then consider how often developers believe a code sample contains multiple predominant paradigms. In Table 7 we show this for each paradigm/code length pair. We see that most participants think the code has multiple *predominant* paradigms. In the cases of Functional and Procedural, this may help to explain some of the confusion we see in the medium-length Procedural code. In particular, it shows that while participants did not think that it definitely has multiple predominant paradigms, their understanding of the difference between the two may be flawed.

With respect to the confidence ratings, We find that there does not appear to be a relationship between confidence and the presented paradigm ($p = .57$). There also does not appear to be a relationship between correctness and confidence ($p = .31$). These both make sense, as we see that in 88% of cases, participants were confident in their answers.

Finding 2: Participants felt very confident in their ability to classify the predominant paradigm, regardless of whether they were correct or not.

6.2.2 Classification Efficiency

To determine whether or not paradigm and task size have an effect on efficiency (time on task), we ran a within-groups 4×3 ANOVA, a summary of which is shown in

Table 6: Classification of code. Participants were asked: "What percentage of the code falls under the following paradigms? The percentages do not have to add up to 100%." Rows represent answers for each presented paradigm. Highlighted cells are high values for each presented paradigm/responded paradigm pair.

	Functional				**Object-Oriented**				**Procedural**			
	<25%	**25–50%**	**51–75%**	**>75%**	**<25%**	**25–50%**	**51–75%**	**>75%**	**<25%**	**25–50%**	**51–75%**	**>75%**
Func.	3	4	5	**17**	**16**	12	1	0	8	**9**	3	**9**
OO	**13**	9	7	0	1	3	5	**20**	8	**9**	8	4
Proc.	6	**9**	5	**9**	**17**	6	3	3	3	6	6	**14**
Mixed	**10**	**10**	5	4	5	8	**10**	6	7	6	3	**13**

Table 7: Do participants perceive multiple paradigms? Participants were asked "Do you think the code has more than one predominant paradigm?" Highlighted cells represent high values for each column.

	Functional			**Mixed**			**Object-Oriented**			**Procedural**		
	Small	**Med**	**Large**	**Small**	**Med**	**Large**	**Small**	**Med**	**Large**	**Small**	**Med**	**Large**
No	**6**	**4**	4	2	2	0	3	4	1	3	**5**	0
Maybe	1	2	0	0	2	2	1	0	0	0	2	2
Yes	3	**4**	**5**	**7**	**6**	**8**	**6**	**5**	**9**	**7**	3	**7**

Figure 2. These results show that there is a significant difference between presented paradigms ($F(3, 104) = 4.93$, $p = .003$, $\eta_p^2 = 0.12$, $f = 0.38$) as well as a significant interaction between presented paradigm and code size ($F(6, 104) = 3.26$, $p = .01$, $\eta_p^2 = 0.16$, $f = 0.43$). Post-hoc comparison of the presented paradigm effect shows a difference between Functional and Object-Oriented ($p = .03$) and Functional and Procedural ($p = .01$); in both cases, the Functional code took longer to classify.

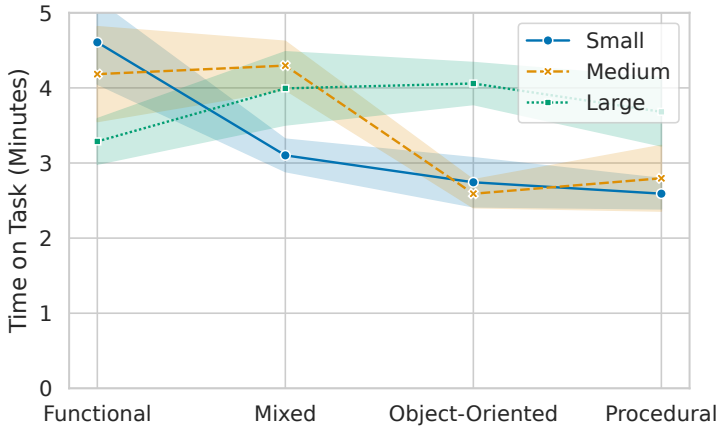

Fig. 2: Average time on task, per paradigm, for classification, including size.

The interaction between paradigm and task size is particularly interesting, though not easy to characterize. When considering the differences highlighted in Figure 2, it appears that the assumed pattern of time-to-completion increasing as code samples became larger does not hold. In fact, the only differences that we see in the interaction are between the Functional Small and Object-Oriented Small ($p = .05$), Functional Small and Procedural Small ($p = .02$), and Functional Small and Object-Oriented Medium ($p = .03$). That is, while we see an interaction, it is only a few particular pairs that we see this on, all of which involve the task that apparently took the longest: Functional Small.

Finding 3: When classifying code, developers take more time when presented with predominantly functional code. In general, however, longer code does not take longer to classify, though the Functional Small took longest.

6.3 RQ2 Results: Language Feature(s) Gazed Upon During Paradigm Classification

We analyze the participants' gazes using both fixation and linearity-based metrics (Table 2).

6.3.1 Classification Fixation Metrics

To determine which specific Python token-level language feature(s) participants look at when classifying predominant paradigms, we first looked into the eye-tracking fixation metrics (summarized in Table 9) and how they differ between the different paradigms and code lengths in the classification task. We ran a within-groups 4×3 ANOVA for fixation count (shown in Table 10), fixation duration, and normalized mean fixation duration. The ANOVAs were conducted on the influence of paradigm and code length on the fixation metrics. For fixation count, all effects were statistically significant except for paradigm. We find that the code length has a significant effect on fixation count ($F(2, 104) = 6.20$, $p = .003$, $\eta_p^2 = 0.11$, $f = 0.35$), and that the interaction effect between the paradigm and code length was significant as well ($F(6, 104) = 2.53$, $p = .03$, $\eta_p^2 = 0.13$, $f = 0.38$). Based on the pairwise comparisons within paradigms and code length (available in the replication package Flint et al. (2023)), it is likely that the code length affects the fixation count more than the paradigm. However, this relationship is not as expected: more fixations did not occur on longer code.

Additionally, we found that while our participants attended to paradigm tokens, it was less frequently than non-paradigm tokens. The results of a t-test and mean fixation counts on paradigm/non-paradigm tokens are shown in Table 8.

We do not see any significant effect on fixation duration from paradigms or code length. However, the results of ANOVA for normalized mean fixation duration show significant interactions between the paradigm and code length ($F(6, 104) = 3.77$, $p = .002$, $\eta_p^2 = 0.18$, $f = 0.47$) and a significant difference between different code lengths ($F(2, 104) = 5.99$, $p = .003$, $\eta_p^2 = 0.10$, $f = 0.34$). These results indicate that code length contributed to more of the difference in normalized mean fixation duration than paradigm, similar to what we see in fixation count.

Table 8: Fixation counts on paradigm and non-paradigm tokens for code classification.

Paradigm	Paradigm Tokens	Non-Paradigm Tokens	t	df	p	p (adj)
Functional	19.07	94.45	-6.78	34.28	$< .001$	$< .001$
Mixed	30.72	87.66	-6.88	55.97	$< .001$	$< .001$
Object-Oriented	40.55	75.76	-3.63	39.21	$< .001$	$< .001$
Procedural	23.29	93.90	-5.53	32.96	$< .001$	$< .001$

Table 9: High-level summary of fixation metrics for classification tasks (ms is milliseconds).

	mean	std
Total Duration (s)	210.34	82.26
Total Duration (m)	3.51	1.37
Fixation Count	115.82	64.18
Fixation Duration (ms)	103,569.76	45,914.62
Mean Fixation Duration (ms)	1,013.35	456.67
Normalized Fixation Duration (ms)	27,365.56	16,286.45
Normalized Mean Fixation Duration (ms)	259.49	131.58
Total Fixations	115.82	64.18
Vertical Next Text (%)	0.31	0.09
Vertical Later Text (%)	0.59	0.06
Horizontal Later Text (%)	0.09	0.04
Regression Rate (%)	0.47	0.03
Saccade Length (px)	167.83	30.55

Table 10: ANOVA Results for Fixation Counts on Classification Tasks

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
paradigm	3	584.58	194.86	0.05	0.9833
variant	2	44567.63	22283.82	6.20	0.0029
paradigm:variant	6	54585.55	9097.59	2.53	0.0251
Residuals	104	373905.43	3595.24		

Finding 4: Code length, not the paradigm, affects the fixation differences when participants classify code.

6.3.2 Visualization of Classification Fixation Metrics

To visualize the differences in fixations, we created scarf plots showing the fixation duration over different token types (related to language features). We show four different scarf plots, each representative of the scarf plots of one paradigm, in Figures 3 to 6. All twelve scarf plots are included in the replication package (Flint et al., 2023). The scarf plots' legends show the token types found in the corresponding code. Token types relevant to the paradigm are specified with vivid colors, whereas the colors of the token types not relevant to the paradigm are faded.

An overview of the scarf plots shows that most participants fixate on relevant token types while reading the code, but they do not show an obvious pattern of (in)correctness being related to fixations on relevant tokens. It can be seen in Fig-

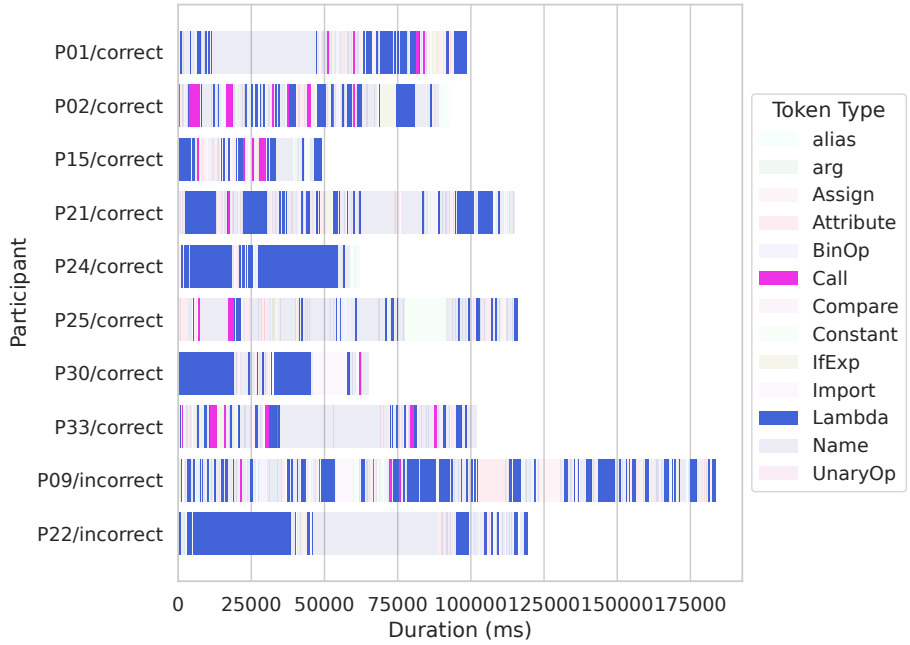

Fig. 3: Scarf plots showing fixations over tokens for code classification task, with participants labeled on the y-axis (participant number and correctness) and time (in ms) on the x-axis: Functional paradigm, Small code length

ures 3 to 5 that the participants who answered incorrectly did spend some time fixating on the paradigm-relevant tokens.

We observe that participants fixated more frequently on `Lambda` tokens, both in the Small (Figure 3) and Large code lengths (see the replication (Flint et al., 2023)) of the Functional paradigm. For the Object-Oriented paradigm, we observe more frequent fixations on the `FunctionDef` token type for all different code lengths and prominence of fixations on `ClassDef` in the Medium code length. Surprisingly, we don't see a pattern in fixating on `ClassDef` in all different code lengths. For the Procedural paradigm, we observe that in both the Large and Small code lengths, most of the participants fixate on `Return, arg, FunctionDef` frequently while reading. This is not the case for `Return` in the Medium code length, which could be because there is only one return statement, and it is placed at the end of the code. As for the mixed paradigm, we did not notice clear patterns of what token types were fixated on more frequently, as the participants looked at different relevant and irrelevant token types for each code length. While in Figure 6, the Large Mixed paradigm code, we see participants fixating mostly on relevant token types during their reading, we do not see similar behavior when looking at the Medium and Small code length plots, which show fixations on both relevant and irrelevant tokens.

Our observations indicate that when participants read code for classifying the predominant paradigm, they fixate on all types of tokens and are not specifically looking for specific token types. In the Functional paradigm, the token types that were fix-

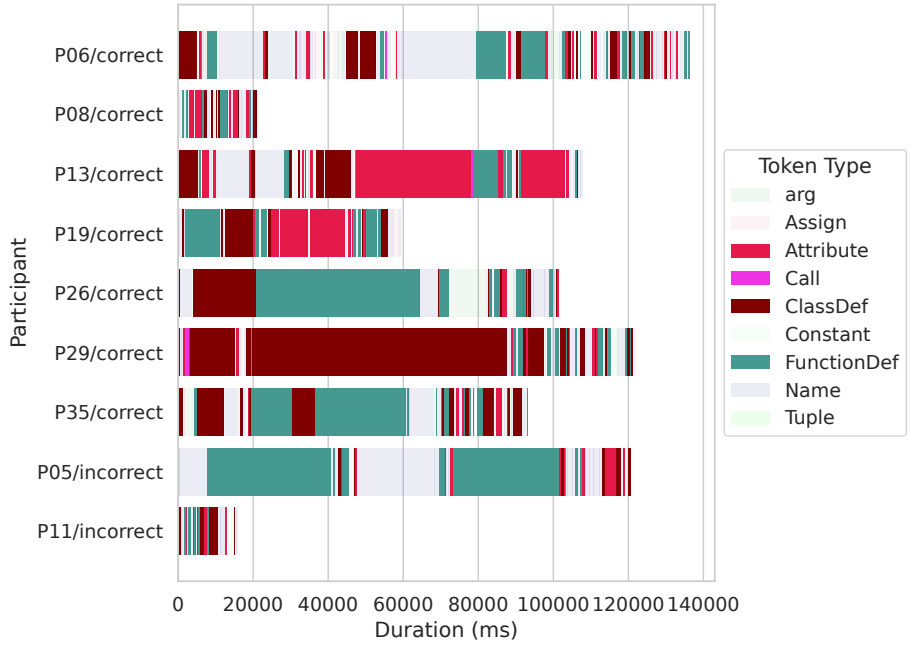

Fig. 4: Scarf plots showing fixations over tokens for code classification task, with participants labeled on the y-axis (participant number and correctness) and time (in ms) on the x-axis: Object-Oriented paradigm, Medium code length

ated on the longest are `Name` and `Lambda`. In the Object-Oriented paradigm, `Name` was fixated on the longest, followed by `Attribute` and `ClassDef`. `Name` and `FunctionDef` were fixated on the longest in the Procedural paradigm, and finally, `Name`, `Constant`, and `FunctionDef` were fixated on the longest in the Mixed paradigm. `Name` was the token type fixated on the longest over all the paradigms and code lengths. The token types the participants focused on seemed to depend on the code itself, not necessarily the paradigm.

Finding 5: When participants read code to classify the predominant paradigm, they fixate on all token types and the tokens they fixate on appear to vary.

6.3.3 Classification Linearity Metrics

Next, we look into the gaze-based linearity metrics, which measure the level of linearity in reading source code. We explore the effects of paradigm and code length on the participants' reading patterns. We ran ANOVAs on the following metrics: vertical reading metrics (*vertical next text*, *vertical later text*), *horizontal later text*, *regression rate*, and *saccade length*. The detailed ANOVA tables are all included in the replication package Flint et al. (2023).

The ANOVA results on vertical reading metrics indicate that the code length and the interaction between paradigm and code length significantly affect the vertical

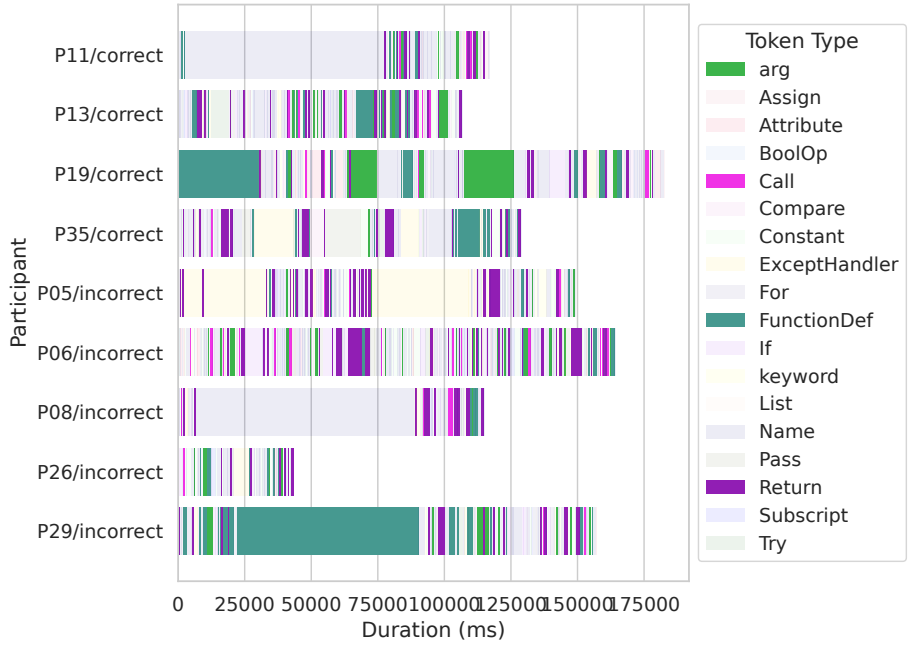

Fig. 5: Scarf plots showing fixations over tokens for code classification task, with participants labeled on the y-axis (participant number and correctness) and time (in ms) on the x-axis: Procedural paradigm, Large code length

code reading patterns. Further observations into the *vertical next text* metric over the code length show that gazes are less likely to move forward one line in the Large code length (Large < Medium < Small) and results from the *vertical later text* metric show that gazes are less likely to move forward multiple lines in the Medium code length and more likely in the Small code length. We observe that both vertical linearity reading metrics are significantly affected by the size of the task code, but not as much by the paradigm.

We look into the horizontal reading patterns by running ANOVA for *horizontal later text* and find that paradigm, code length, and the interaction between paradigm and code length significantly affect the horizontal reading patterns. The results show that gazes are less likely to move forward within a line in the Procedural paradigm but more likely to do so in Small code lengths. We could not find significant differences between the groups looking at the *regression rate* metric, which indicates that there are not significant differences between how gazes move backward horizontally on a line in the different paradigms and code lengths studied.

Finding 6: The linearity metrics show that code length (rather than paradigm) is important in how developers read code. The vertical linearity metrics significantly differ for different code lengths, indicating different vertical movement searching patterns during tasks.

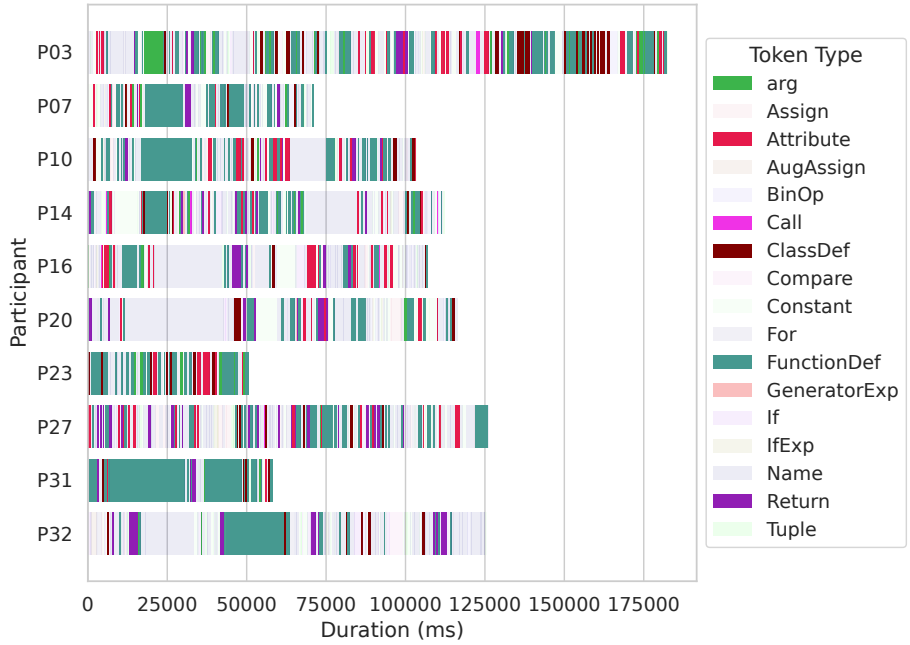

Fig. 6: Scarf plots showing fixations over tokens for code classification task, with participants labeled on the y-axis (participant number and correctness) and time (in ms) on the x-axis: Mixed paradigm, Large code length

6.4 RQ3 Results: Accuracy and Efficiency of Debugging Logical Errors

Next, we examine RQ3, asking if the predominant paradigm has an effect on the ability to debug logical errors. For this, we consider two metrics: accuracy and time-to-completion.

6.4.1 Debugging Accuracy

We show the judgment accuracy broken down by both paradigm and program in Table 11. We find that the paradigm of the code does not have a statistically significant effect on participants' ability to determine which line contains the bug ($Q(3) = 7.74$, $p = .05$, $\eta^2_Q = 0.09$). However, as $p = .05$, we perform pair-wise comparison anyway. Bonferonni-adjusted p-values show no difference between paradigms, though there may exist a difference ($\chi^2(1) = 4.90$, $p_{adj} = .16$, $p = .03$) between Procedural (93% accurate) and Mixed (66% accurate). That is, the paradigm does not influence the ability to debug code.

However, the program does appear to influence correctness ($Q(3) = 11.95$, $p = .01$, $\eta^2_Q = 0.14$), though after pair-wise comparison we only suspect a difference between Factorial and Cube ($\chi^2(1) = 6.75$ $p_{adj} = .06$) and between Factorial and Palindrome ($\chi^2(1) = 4.08$, $p_{adj} = .26$, $p = .04$). It appears no matter the

paradigm, participants may have found the Factorial code more difficult to debug (59% accurate).

Table 11: Number of correct responses to number of participants who viewed a given stimulus in bug localization. Highlighted cells represent stimuli categories that are different from the whole.

	Cube	Factorial	Largest Number	Palindrome	Overall
Functional	7 / 7	4 / 8	7 / 8	6 / 6	82.76%
Mixed	5 / 6	4 / 7	4 / 8	6 / 8	**65.52%**
Object-Oriented	7 / 8	3 / 6	5 / 7	6 / 8	72.41%
Procedural	8 / 8	6 / 8	6 / 6	7 / 7	**93.10%**
Overall	93.10%	**58.62%**	75.86%	86.21%	

Finally, we examine the relationships between confidence in the answer and presented paradigm, between confidence and program, and confidence and correctness. Participants appeared less confident than when classifying code (64% at least "Confident", compared to 88%). There appears to be a statistical relationship between confidence and whether a participant could select the correct line ($p < .001$). We also see a relationship between the program and confidence ($p = .002$), suggesting that participants find some programs more difficult. We do not see a relationship between the presented paradigm and confidence ($p = .20$).

Finding 7: The code's paradigm may influence how accurately developers can debug, with accuracy higher in Functional and Procedural code. Additionally, no difference in confidence between the different paradigms was found, though participants were more confident in some programs than others.

6.4.2 Debugging Efficiency

In Figure 7 (above), we again present the results of a factorial ANOVA (4×4) to explore the effects of paradigm and program on time to debug. In this case, we see significant effects for the presented paradigm ($F(3, 100) = 4$, $p = .01$, $\eta_p^2 = 0.11$, $f = 0.35$), program ($F(3, 100) = 8.45$, $p < .001$, $\eta_p^2 = 0.16$, $f = 0.50$), and the interaction of the two. We again perform pairwise comparisons.

We see a significant difference between Functional and Mixed paradigm code ($p = .03$), and between Functional and Procedural code ($p = .01$); in both cases, we see developers took longer to find the bug in Functional code. We also see a difference between Factorial and Largest Number ($p = .03$), as well as Factorial and Cube ($p < .001$), and Palindrome and Cube ($p = .002$).

When we consider the interaction of paradigm and program, we notice that Functional+Factorial takes the longest and is different even from the Mixed+Factorial programs (complete results of Tukey's HSD are available in the replication package Flint et al. (2023)). This interaction follows the patterns we see for paradigm and program and may drive both effects.

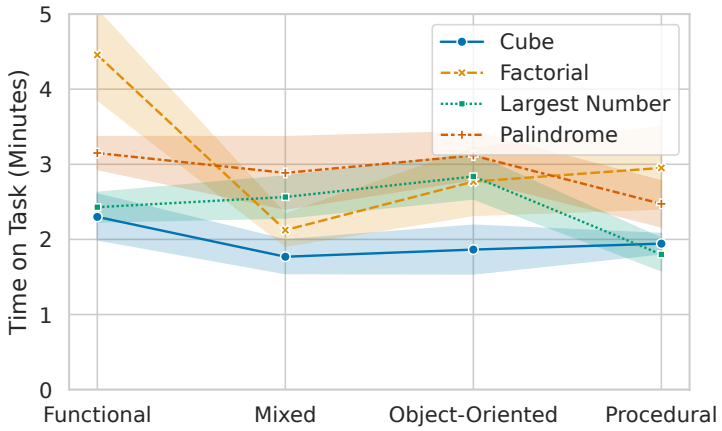

Fig. 7: Average time on task, per paradigm, for debugging, including program.

Finding 8: Functional code takes significantly longer to debug than Mixed and Procedural.

6.5 RQ4 Results: Language Feature(s) Gazed Upon When Debugging Logical Errors

6.5.1 Debugging Fixation Metrics

To determine which specific Python language feature(s) developers look at when debugging, we look at the influence of the paradigm and program on finding the bug by running a factorial 4×4 ANOVA for fixation count (shown in Table 14), fixation duration, and normalized mean fixation duration metrics (summarized in Table 13). For fixation count, the ANOVA results show a significant effect for paradigm ($F(3, 100) = 7.79$, $p < .001$, $\eta_p^2 = 0.19$, $f = 0.48$), program ($F(3, 100) = 13.55$, $p < .001$, $\eta_p^2 = 0.29$, $f = 0.64$), and the interactions of the two ($F(9, 100) = 4.22$, $p < .001$, $\eta_p^2 = 0.28$, $f = 0.62$). Post-hoc comparison (detailed in the replication package Flint et al. (2023)) shows that Procedural and Functional paradigms are the driving factors behind the significant differences in fixation count. The tasks in the Functional paradigm had the highest fixation count, followed by Object-Oriented, Mixed, and Procedural. Cube is the program that affects the fixation count the most among other programs, with the fixation count for the program being the lowest among all paradigms. This could have also been affected by the small length of the Cube program.

Similar to our findings for code classification, we found that while our participants attended to paradigm tokens during bug localization, it was less frequently than non-paradigm tokens. The results of a t-test and mean fixation counts on paradigm/non-paradigm tokens are shown in Table 12.

Next, we investigate the effects of paradigm and program on fixation duration. Once again, we see significant differences in fixation duration between dif-

Table 12: Fixation counts on paradigm and non-paradigm tokens for bug localization.

Paradigm	Paradigm Tokens	Non-Paradigm Tokens	t	df	p	p (adj)
Functional	22.57	116.41	-5.31	29.57	< .001	< .001
Mixed	22.73	73.52	-6.19	42.34	< .001	< .001
Object-Oriented	42.45	75.48	-3.62	51.38	< .001	< .001
Procedural	18.86	54.21	-6.35	34.53	< .001	< .001

Table 13: High-level summary of fixation metrics for bug localization tasks (ms is milliseconds).

	mean	std
Total Duration (s)	156.61	67.29
Total Duration (m)	2.62	1.12
Fixation Count	105.78	72.51
Fixation Duration (ms)	104,978.53	55,167.70
Mean Fixation Duration (ms)	1,244.51	1,163.48
Normalized Fixation Duration (ms)	37,299.86	22,706.22
Normalized Mean Fixation Duration (ms)	493.75	853.02
Total Fixations	105.78	72.51
Vertical Next Text (%)	0.56	0.13
Vertical Later Text (%)	0.72	0.08
Horizontal Later Text (%)	0.21	0.07
Regression Rate (%)	0.47	0.03
Saccade Length (px)	117.99	39.95

Table 14: ANOVA Results for Fixation Counts on Localization Tasks.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
paradigm	3	69901.00	23300.33	7.78	0.0001
variant	3	121635.06	40545.02	13.55	0.0000
paradigm:variant	9	113780.19	12642.24	4.22	0.0001
Residuals	100	299309.93	2993.10		

ferent paradigms ($F(3, 100) = 3.02$, $p = .03$, $\eta_p^2 = 0.08$, $f = 0.30$), programs ($F(3, 100) = 8.49$, $p < .001$, $\eta_p^2 = 0.20$, $f = 0.50$), and the interactions of the two ($F(9, 100) = 2.13$, $p = .03$, $\eta_p^2 = 0.16$, $f = 0.44$). Post-hoc comparison (detailed in the replication package Flint et al. (2023)) showed significant differences between Functional and Procedural programs, with the Functional paradigm having the highest fixation duration among all paradigms, similar to the pattern of fixation count. Once again, similar to the fixation count metric, the Cube program drives the differences between programs.

The final fixation-based metric we look at is normalized mean fixation duration. We see a significant effect from the programs ($F(3, 100) = 4.57$, $p = .005$, $\eta_p^2 = 0.12$, $f = 0.37$) and the interaction between programs and paradigm ($F(9, 100) = 2$, $p = .05$, $\eta_p^2 = 0.15$, $f = 0.42$). A post-hoc comparison shows a similar pattern to fixation count and duration, with Cube affecting the metric significantly. On average, normalized mean fixation duration was higher in the Cube program, specifically in the Functional paradigm, in which this metric was the highest among all tasks and programs. The results indicate that although the Cube program is small with the lowest fixation count and duration among all paradigms, it still required rela-

tively longer focus and more cognitive load from the participants. Additionally, even though the Factorial program had the highest fixation count and duration, its normalized mean fixation duration was, on average, similar to the Largest Number and Palindrome, further confirming the effect of the Cube program on the differences.

Finding 9: Fixation-based metrics for the debugging tasks indicate that paradigm and program significantly influence the metrics. The Functional code has the largest effect on metrics, as does the Cube program; notably the Functional Cube task required the longest focus from the participants

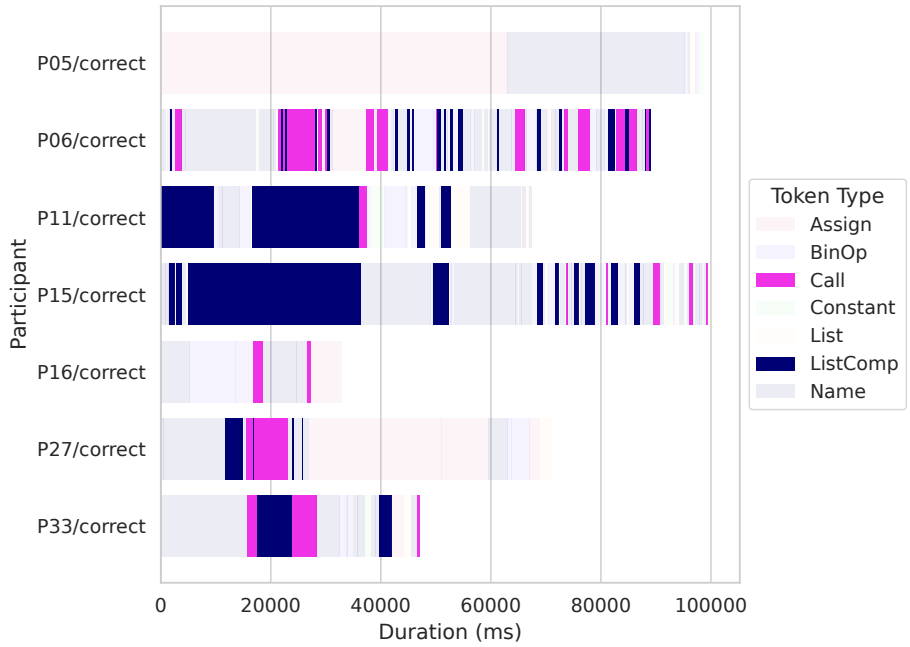

Fig. 8: Scarf plots showing fixations over tokens for bug localization task, with participants labeled on y-axis (participant number and correctness) and time (in ms) on the x-axis: Functional paradigm, Cube program

6.5.2 Visualization of Debugging Fixation Metrics

To visualize the differences in fixations, we once again use scarf plots showing in Figures 8 to 11 the fixation duration over different token types, the remaining twelve are available in our replication package (Flint et al., 2023). Post-hoc pairwise comparison of the fixation metrics showed us significant differences between the Functional and Object-Oriented paradigms of the Cube program and the Functional and Procedural paradigms of the Factorial program. First, we look at how participants fixated over tokens while solving the Cube task in the Functional paradigm (Figure 8) versus

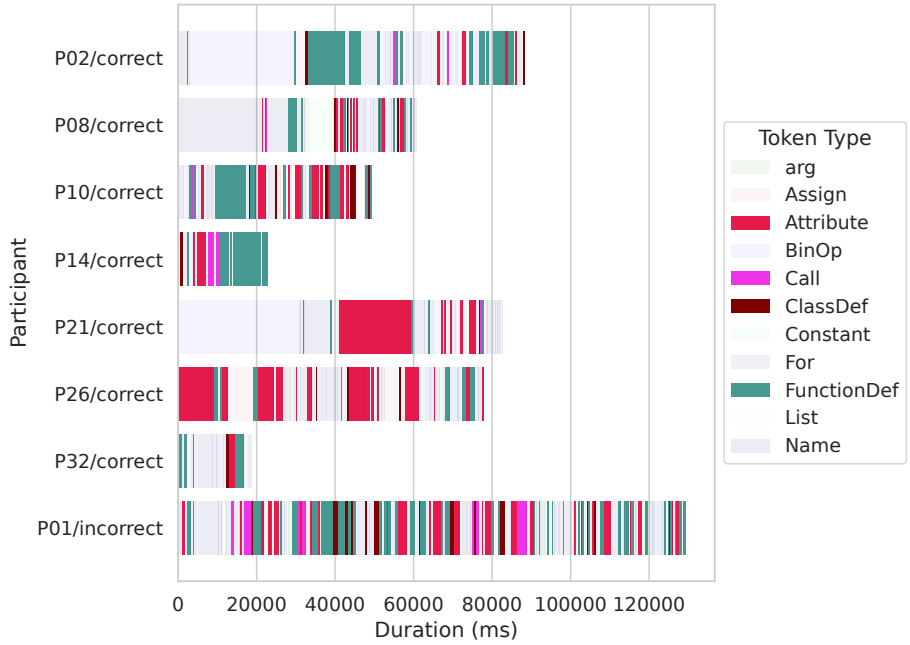

Fig. 9: Scarf plots showing fixations over tokens for bug localization task, with participants labeled on y-axis (participant number and correctness) and time (in ms) on the x-axis: Object-Oriented paradigm, Cube program

the Object-Oriented paradigm (Figure 9). We notice that while the participants in both groups have fixated on a lot of paradigm-relevant token types, the fixations on these tokens are relatively longer in the Functional paradigm. The longer fixations indicate that participants needed more time and focus to understand the Functional program of Cube and find the error.

Next, we look at the scarf plots for the Functional and Procedural paradigms of the Factorial program. In the Functional (Figure 10) paradigm, while the participants fixated on the paradigm-relevant token types frequently, they did not fixate on these tokens longer than the non-relevant tokens (this can be explained by this task's wider variety of token types, compared to the Cube program). In the Procedural (Figure 11) paradigm, despite the wider variety of token types, participants spent more time focusing on the paradigm-relevant token types, especially `FunctionDef` and `Return`.

Finally, we look at what specific tokens were fixated on during debugging. Except in the case of Procedural, `Name` tokens were fixated upon the most (in Procedural, `JoinedString` was fixated on more). In the Functional code, developers fixated upon `ListComp` fourth most often, though `FunctionDef` was fixated upon second most often. In Object-Oriented, developers fixated upon `Attribute` second most often, with no other paradigm-specific tokens found in the top five. Procedural code saw `FunctionDef` and `Assign` as the fourth and fifth most common, respectively.

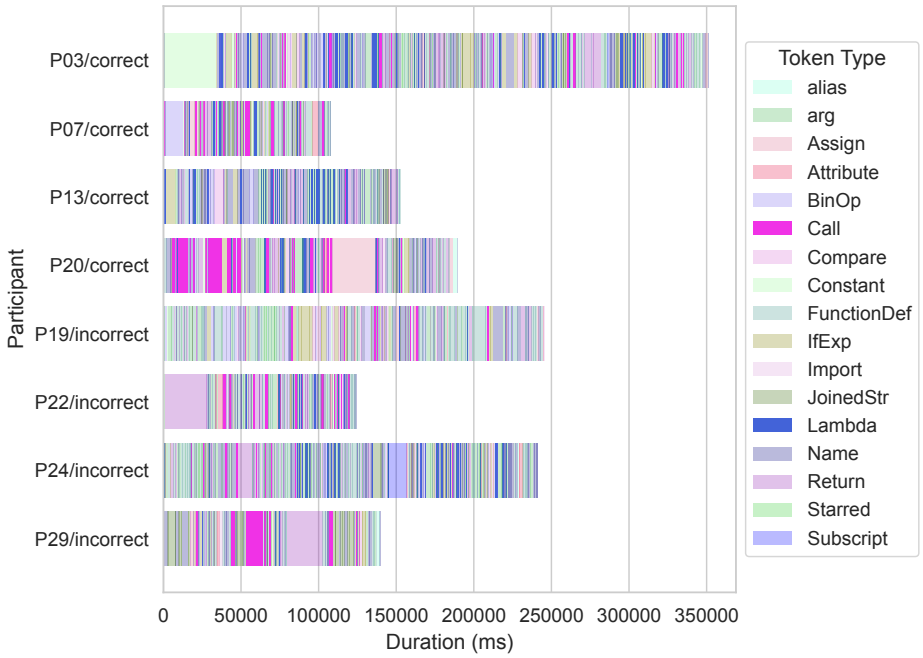

Fig. 10: Scarf plots showing fixations over tokens for bug localization task, with participants labeled on y-axis (participant number and correctness) and time (in ms) on the x-axis: Functional paradigm, Factorial program

Finding 10: We found that both paradigm and program significantly influence fixation count and duration for debugging tasks, highlighting their importance in debugging.

6.5.3 Debugging Linearity Metrics

We looked into the gaze-based linearity metrics to understand how paradigm and program affect them during debugging. We ran ANOVAs on the vertical reading metrics (*vertical next text* and *vertical later text*) and found that paradigm, program, and their interaction significantly affect these reading patterns. Specifically, by looking at the post-hoc comparison, we observe that these significant differences are driven by the Functional paradigm and the Cube program, which is similar to our findings with fixation metrics. Both metrics show that gazes are most likely to move forward one or multiple lines in the Functional paradigm (Functional > Procedural > Mixed > Object-Oriented). The *horizontal later text* follows the same pattern as the vertical metrics. While the *saccade length* is largest in the Functional paradigm, it's smallest in the Procedural paradigm (Functional > Object-Oriented > Mixed > Procedural). Finally, the *regression rate* metric is affected only by the paradigm, with significant differences between the Functional paradigm and other paradigms, indicating that gazes are less likely to move backward in the Functional paradigm

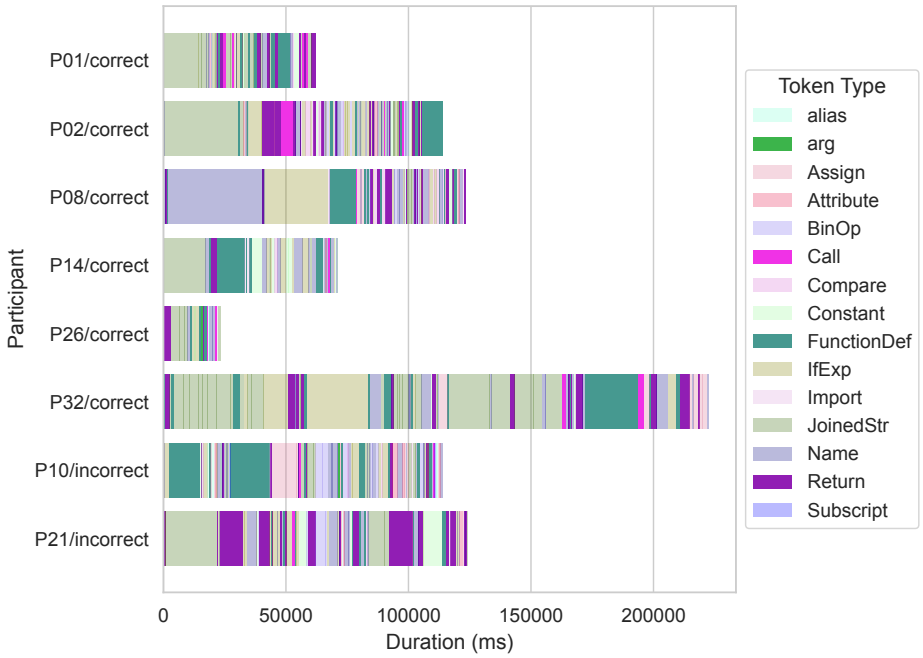

Fig. 11: Scarf plots showing fixations over tokens for bug localization task, with participants labeled on y-axis (participant number and correctness) and time (in ms) on the x-axis: Procedural paradigm, Factorial program

code. The detailed ANOVA tables are all included in the replication Flint et al. (2023).

Finding 11: For debugging tasks, gaze-based linearity metrics, particularly vertical next text and vertical later text, are significantly influenced by paradigm and program, with the Functional Cube program showing distinct reading behavior, characterized by forward movements. In particular, the Functional paradigm showed many single-line and multi-line forward movements.

7 Discussion

This study was conducted on two types of tasks on Python code — paradigm classification and debugging within a specific code paradigm in Python. Our results show interesting differences in reading behavior and performance for the Functional code across both the paradigm classification and bug localization tasks. In the next section, we first interpret our results for each research question. These differences in reading behavior and differences noted more generally, between Functional code and code in other paradigms, have some interesting downstream impacts, particularly on theory building, education, research, and practice. The rest of the section discusses these impacts.

7.1 Interpretation of Results

The study investigated four research questions that resulted in eleven unique findings as described in the results. Below, we seek to distill some high level observations from those findings. First, we note that we always distinguish task (classification or debugging) and stimuli (the specific program given to developers). This said, we offer the following explanations for our results.

When we consider our findings for RQ1 and RQ2 (Findings 1 to 6), we see a potential explanation. The task of code classification does not require a particularly in-depth understanding of code; instead, the driving factor is to locate paradigm-specific tokens, even if they are not attended as frequently as non-paradigm tokens (Finding 5) The drive to detect paradigm-specific tokens may also explain why certain paradigms are easier to classify, as participants may have been more familiar with certain paradigms than others (as they are more common or have more memorable keywords (Finding 1)) or certain paradigms might structure the code in a manner that is more familiar to participants (such as Object-Oriented code having a standard structure starting with a class declaration and then indented methods). This may also explain the differences in reading linearity driven by length (Finding 6). This same knowledge issue may also explain why some paradigms took longer to classify (Finding 3), however, the generally uniform confidence of the participants (Finding 2) may be instead due to feeling comfortable with the classification schema.

During debugging, it is unsurprising that paradigm influences reading linearity (Findings 9 to 11): paradigm influences how code is written, particularly in a language like Python, where definition order is important. Because localizing a bug requires a deeper understanding, it is generally necessary to follow a program's logic, often following the data or control flow, which may more or less be top-down in different paradigms. This is especially notable for Functional code, which exhibited a large amount of forward movement, following very linear code. The higher fixation counts, longer fixation durations, and high vertical and horizontal reading patterns are indicative of high cognitive load.

Our results also indicate that procedural code had the highest accuracy for finding bugs (Finding 7). This is indicative of the fact that even though participants knew Python, the functional paradigm in Python was not as familiar to them as the other paradigms, and may also have impacted the code classification tasks. These results should indicate that we need to consider changing how we teach Python. As software engineers, we are often told to use the right tool for the job. Perhaps in some cases, the Functional paradigm usage in Python is the most efficient way to solve a problem, but if it carries with it a high cognitive load (for whatever reason — perhaps lack of knowledge structures for comprehension), it makes it harder for someone without appropriate training to comprehend it.

7.2 Impacts on Education

The observed differences in code classification correctness are especially important to the education field. We saw that participants clearly understood one paradigm (Object-Oriented code), but appeared confused between Functional and Procedural code. This is especially notable as, in Python, only 40% of files were previously found to be predominantly Object-Oriented (Dyer and Chauhan, 2022). This in particular

could be caused by a lack of familiarity with these paradigms in general or in how they are exhibited in Python. In either case, explaining the different paradigms and showing their use to students early may help to reduce differences in classification correctness, and over time, may induce changes in how developers classify and read code.

7.3 Impacts on Research

When we consider bug localization, the differences in accuracy between paradigms are lessened, though mixed-paradigm code has the lowest percentage of correct answers. This does not follow with the differences in linearity. Thus, understanding the connection between the reading linearity and the act of debugging will require further research, especially given that some paradigms, such as Functional, involve more and larger movement as well as being uncommon in code. In particular, determining if features of Functional code are atoms of confusion may be particularly helpful (da Costa et al., 2023).

7.4 Impacts on Practice

Finally, because we found that developers tend to perform poorly in Mixed-paradigm code, we believe that developers should be careful in how they mix different paradigms. In particular, it is important that developers make sure the paradigms are well-matched. What that looks like for a given project should be specified in its style guide. However, given our primarily-student population, further research is necessary to determine the effect of mixing paradigms on professional developers. Additional research is likewise necessary to determine what a style guide could look like for mixed paradigm code.

7.5 Impacts Towards Future Theory

The program comprehension literature reviewed by Storey et al. almost two decades ago (Storey, 2006, 2005) presents the main program comprehension theories as presented early in the 1960's when the procedural paradigm was prevalent. These theories include the use of beacons and chunking (Wiedenbeck and Evans, 1986), top-down vs. bottom-up comprehension (Letovsky, 1987; Brooks, 1983), and programming plans (Storey, 2005). These theories suggest that developers use prior knowledge bases and structures to understand programs. In 2017, Siegmund et al. use neuroscience methods (fMRI, in this case) to validate some of these theories and showed beacons ease comprehension but did not find support for programming plans (Siegmund et al., 2017). The reasons for this could be many one being the low expertise of participants. They call for more studies to validate this theory in future work.

We posit that theories such as cognitive load theory (Hollender et al., 2010; Duran et al., 2022; Debue and van de Leemput, 2014) could also be directly related to why the functional paradigm took longer in our study. These theories need to be studied in relation to program comprehension of mixed paradigm code. Gaze

behavior in itself might not be the only way towards building an underlying theory about how developers work with Python code written in mixed paradigms. But it can certainly supplement other methods (surveys, think-aloud, interviewing) of gathering evidence. Gaze gives us fine-grained details of where developers look while solving a task and shows us the thought processes being used to develop their mental model. We view this study as one of the first steps in understanding different paradigm usage in Python.

With the addition of newer paradigms and the use of coding assistants like ChatGPT and Copilot, a fresh look into whether prior theories apply to program comprehension tasks is needed. Program comprehension research should look into whether these existing theories are still valid or need refinement as part of this culture shift. As we gather more evidence in this area of mixed paradigm languages, we help gather evidence towards building a theory of mixed paradigm comprehension and debugging. These theories can also help program language designers as they work on new languages. For example, what can language designers incorporate to make functional languages more user friendly? We are not aware of any theories claimed to be used in programming language design.

We can anticipate a theory that functional programming takes longer since it requires longer time to build the mental structures indicating higher cognitive load (Duran et al., 2022), to understand what the program does. There are many other factors that could also play in role in the way developers comprehend different paradigms related to knowledge structures, prior work experience, and other socio-technical factors. As more eye tracking studies are conducted with Python, researchers will be able to extract common phenomena observed in developer behavior that can eventually be used to formulate a theory of program comprehension for mixed paradigm languages such as Python. Such a theory would involve many factors including the developers behavior, processes used, paradigms, experience, prior knowledge, project domain and more.

8 Threats to Validity

We discuss potential threats to the validity of our study and how we mitigate them.

Construct Validity. First is the threat of start/stop time measurement. These measurements are derived from when the participant first starts looking at the target file and when they stop reading it. There may be a small lag in stop times due to the manual nature of stopping recording upon participant completion. As this lag is fairly consistent for all tasks, we do not believe it affects the results. How the paradigm is defined bears some influence on responses as participants may have a different internal definition of a paradigm and the definition used may be confusing. This is especially notable as participants' prior experience potentially informs their internal definitions so that prior experience may have biased their answers. Our background survey failed to collect data on participant experience with each paradigm, and results may be biased. Moreover, as different syntactic features may co-occur, this may cause other problems. We remedy this by consistently using definitions from prior literature Dyer and Chauhan (2022). Additionally, we used a tutorial to train participants in the definitions of the language features used. The code presented is

fairly linear, which may not accurately reflect how all paradigms are used in the real world.

We use fixation counts and fixation durations for our eye tracking metrics for RQ2 and RQ4. An alternate method could use the fixations per second (fixation rate) metric for fixation counts. The number of fixation counts are inherently dependent on the time on task, especially since the time is different for each participant (per task) as in our study. We mitigate this risk by using fixation durations (also related to task time) and normalizing them by dividing by the length of the fixated-on token.

Finally, the study was set up to provide good calibration on the presented code (accessible font size) using a high-quality research-grade tracker. All the tasks were less than 5 mins in length with recalibrations done between tasks. Each task was also recorded separately. In addition, the moderator made sure the tracking was done accurately by watching a live view of the participants' head position and gently prompting them to adjust when needed. Because of the above stated reasons, we did not find the need to correct fixations. Automatically correcting fixations does require a validated golden set as shown by Guarnera et al. (2025) for Java source code.

Internal Validity. There are several threats to internal validity. The first threat is our fixed order of presentation for classification. Classification tasks were always presented as "Functional", "Object-Oriented", "Procedural", then "Mixed". This fixed order may have introduced two possible threats: fixed order may confound the relationship to paradigm, and fixed order may have induced a learning effect. This is a particular threat to our conclusions on correctness and time to completion on classification. However, because of the fixed order of task, and un-fixed order of paradigm presentation in bug localization, we can confirm at least some of these differences, especially as relates to eye-tracking-derived metrics.

Finally, our explicit training on the paradigm classification presents a threat to validity. Because we explicitly train participants, this can lead to a mismatch and bias on the measures related to correctness or time-on-task. However, because of how the training is completed, we do not believe that the eye-tracking-specific measures are likely to be impacted.

External Validity. Our study had primarily student participants, limiting generalizability. In particular, students' limited experience with multi-paradigm code, or any given paradigm, limits the ability to generalize to developers with broader experience. However, as this study is primarily exploratory in nature, the use of convenience sampling is sound.

Another threat concerns the tasks themselves. As the tasks used were short and simple in terms of complexity, the ability to generalize to more complex code may be at stake. However, despite the simplicity, this limitation may matter less than expected as we see a number of differences. The tasks are all written in Python, so results may not generalize to other multi-paradigm languages.

Moreover, the small number of tasks further reduces generalizability, as relatively few paradigm-related constructs were studied. However, the small number of tasks increases both internal and conclusion validity, allowing us to better compare results across paradigms.

Additionally, the varying sizes (and designs) of the tasks across paradigms reduces our ability to truly generalize the results with respect to difficulty or time.

However, the subjective results of perceived difficulty should remain a strong indicator. Additionally, although there is variance in task size, all tasks remain quite small, which improves comparability but does reduce generalizability to larger tasks.

Conclusion Validity. Our statistical analyses are well-supported by our experimental design. First, since the variables analyzed are binary (correctness), mutually exclusive, matched (or paired in the case of the χ^2) and independent per subject, the use of the non-parametric Q with McNemar's χ^2 to follow-up is supported. Moreover, our use of Bonferroni adjustment reduces the risk of Type I error in this circumstance.

Next, our use of Fisher's exact test to analyze relationships between confidence and the presented paradigm, length, or program is similarly appropriate since the data analyzed is categorical (and in some cases ordered), with small, fixed marginal totals.

Additionally, our sample size (at 28 retained participants) is small. However, for an observed effect size of $f = 0.30$, (fixation duration on localization, difference between paradigm), we have a theoretical power of 74.7%. Although 74.7% power is not ideal, other effect sizes are higher, and at $f = 0.35$, we have 87.7% power, which is substantially better. That is, we generally have at least sufficient power to detect our effects.

Finally, our use of Factorial Mixed-Groups ANOVA is supported since our dependent variables are continuous and grouping variables are as required (i.e., within-groups variables are truly within-groups, and between-groups variables are truly between-groups). We do not see any outliers in the data, and mixed-group models are known to be robust to minor deviations from normality. As Tukey's HSD requires similar assumptions, it is likewise supported. There may, however, be other confounding factors, such as the size of the code across paradigms.

9 Conclusions and Future Work

In this study, we used eye tracking to first determine if developers look at particular language features when attempting to classify the predominant paradigm in Python code, and second, if language features played a role in isolating bugs. We found that developers tend to take longer to classify code written in the functional paradigm, but their confidence does not change based on the shown paradigm. Developers also incorrectly classified procedural code, likely because they confused procedural and functional constructs. When classifying code, we found that participants fixated on all types of tokens, not only those specific to the paradigm of code they were shown. Moreover, we found that paradigm does not strongly influence developers' ability to isolate bugs, though participants were overall less confident. One paradigm in particular (Functional) did cause participants to take longer to isolate bugs. However, the program they were debugging also influenced this. Finally, fixation-derived metrics (including linearity) were affected by Functional code which was read in a more linear order than other paradigms.

In the future, we want to better understand why developers consider code as being mixed and not predominantly in one paradigm. Additionally, we want to explore the connection between atoms of confusion, code smells, and programming paradigms.

Data Availability

All de-identified data and processing scripts are available in our replication package Flint et al. (2023) on Zenodo.

Compliance with Ethical Standards

Conflict of Interest

The authors declare that they have no conflict of interest.

Human Participants and Informed Consent

This experiment was reviewed and approved by the University of Nebraska–Lincoln Institutional Review Board, as project #21455, IRB number #20220121455EX. Study participants gave informed consent and were able to withdraw consent at any time.

Funding

This material is based upon work supported by the U.S. National Science Foundation under grant numbers 23-46327 and CCF 18-55756.

Acknowledgements We thank all the participants of this study for their time.

References

Abid NJ, Sharif B, Dragan N, Alrasheed H, Maletic JI (2019) Developer reading behavior while summarizing Java methods: Size and context matters. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, IEEE Press, New York, NY, USA, pp 384–395, URL https://doi.org/10.1109/ICSE.2019.00052

Alexandru CV, Merchante JJ, Panichella S, Proksch S, Gall HC, Robles G (2018) On the usage of Pythonic idioms. In: Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Association for Computing Machinery, New York, NY, USA, Onward! 2018, p 1–11, URL https://doi.org/10.1145/3276954.3276960

Andersson R, Larsson L, Holmqvist K, Stridh M, Nyström M (2017) One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms. Behavior research methods 49(2):616–637, URL https://doi.org/10.3758/s13428-016-0738-9

Beelders T, du Plessis JP (2016) The influence of syntax highlighting on scanning and reading behaviour for source code. In: Proceedings of the annual conference of the South African institute of computer scientists and information technologists, Association for Computing Machinery, New York, NY, USA, pp 1–10, URL https://doi.org/10.1145/2987491.2987536

Behler J, Weston P, Guarnera DT, Sharif B, Maletic JI (2023) iTrace-Toolkit: A pipeline for analyzing eye-tracking data of software engineering studies. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE Press, New York, NY, USA, pp 46–50, URL https://doi.org/10.1109/ICSE-Companion58688.2023.00022

Ben-Shachar MS, Lüdecke D, Makowski D (2020) effectsize: Estimation of effect size indices and standardized parameters. Journal of Open Source Software 5(56):2815, URL https://doi.org/10.21105/joss.02815

Berger ED, Hollenbeck C, Maj P, Vitek O, Vitek J (2019) On the impact of programming languages on code quality: A reproduction study. ACM Trans Program Lang Syst 41(4), URL https://doi.org/10.1145/3340571

Berry KJ, Johnston JE, Paul W Mielke J (2007) An alternative measure of effect size for Cochran's q test for related proportions. Perceptual and Motor Skills 104(3_suppl):1236–1242, URL https://doi.org/10.2466/pms.104.4.1236-1242

Binkley D, Davis M, Lawrie D, Maletic JI, Morrell C, Sharif B (2013) The impact of identifier style on effort and comprehension. Empirical Software Engineering 18(2):219–276, URL https://doi.org/10.1007/s10664-012-9201-4

Boslaugh S (2013) Statistics in a Nutshell, 2nd edn. O'Reilly Media, Inc.

Brooks R (1983) Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies 18(6):543–554, DOI https://doi.org/10.1016/S0020-7373(83)80031-5, URL https://www.sciencedirect.com/science/article/pii/S0020737383800315

Busjahn T, Schulte C, Sharif B, Simon, Begel A, Hansen M, Bednarik R, Orlov P, Ihantola P, Shchekotova G, et al. (2014) Eye tracking in computing education. In: Proceedings of the 10th annual conference on International computing education research, Association for Computing Machinery, New York, NY, USA, pp 3–10, URL https://doi.org/10.1145/2632320.2632344

Busjahn T, Bednarik R, Begel A, Crosby M, Paterson JH, Schulte C, Sharif B, Tamm S (2015) Eye movements in code reading: Relaxing the linear order. In: 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, IEEE Press, New York, NY, USA, pp 255–265, URL https://doi.org/10.1109/ICPC.2015.36

Carbonnelle P (2023) PYPL PopularitY of Programming Language. https://pypl.github.io/

Chen Z, Chen L, Ma W, Xu B (2016) Detecting code smells in Python programs. In: 2016 International Conference on Software Analysis, Testing and Evolution (SATE), IEEE, New York, NY, USA, pp 18–23, URL https://doi.org/10.1109/SATE.2016.10

Cochran WG (1950) The comparison of percentages in matched samples. Biometrika 37(3–4):256–266, URL https://doi.org/10.1093/biomet/37.3-4.256

da Costa JS, Gheyi R, Castor F, de Oliveira PRF, Ribeiro M, Fonseca B (2023) Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. Empirical Software Engineering 28(81), URL https://doi.org/10.1007/s10664-023-10311-0

Crosby ME, Stelovsky J (1990) How do we read algorithms? A case study. Computer 23(1):24–35, URL https://doi.org/10.1109/2.48797

Debue N, van de Leemput C (2014) What does germane load mean? an empirical contribution to the cognitive load theory. Frontiers in Psychology 5, DOI 10.3389/fpsyg.2014.01099, URL https://www.frontiersin.org/articles/10.3389/fpsyg.2014.01099

Duchowski AT (2017) Eye tracking methodology: Theory and practice. Springer London, London, UK, URL https://doi.org/10.1007/978-1-84628-609-4

Duran R, Zavgorodniaia A, Sorva J (2022) Cognitive load theory in computing education research: A review. ACM Trans Comput Educ 22(4), DOI 10.1145/3483843, URL https://doi.org/10.1145/3483843

Dyer R, Chauhan J (2022) An exploratory study on the predominant programming paradigms in Python code. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, pp 684–695, URL https://doi.org/10.1145/3540250.3549158

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, New York, NY, USA, ICSE '13, p 422–431, URL https://doi.org/10.1109/ICSE.2013.6606588

Eghbali A, Pradel M (2022) Dynapyt: A dynamic analysis framework for Python. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, p 760–771, URL https://doi.org/10.1145/3540250.3549126

Farooq A, Zaytsev V (2021) There is more than one way to zen your Python. In: Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering, Association for Computing Machinery, New York, NY, USA, SLE 2021, p 68–82, URL https://doi.org/10.1145/3486608.3486909

Fisher RA (1922) On the interpretation of χ^2 from contingency tables, and the calculation of p. Journal of the Royal Statistical Society 85(1):87–94, URL https://doi.org/10.2307/2340521

Flint SW, Chauhan J, Mansoor N, Sharif B, Dyer R (2023) Replication package for "An exploratory eye tracking study on how developers classify and debug Python code in different paradigms". URL https://doi.org/10.5281/zenodo.8179167

Floyd RW (1979) The paradigms of programming. Communications of ACM 22(8):455–460, URL https://doi.org/10.1145/359138.359140

Foundation DS (2021) Django: The web framework for perfectionists with deadlines. https://www.djangoproject.com/

Freitas V (2021) Class-based views vs. function-based views. https://simpleisbetterthancomplex.com/article/2017/03/21/class-based-views-vs-function-based-views.html

Github (2022) The top programming languages. https://octoverse.github.com/2022/top-programming-languages

Guarnera DT, Bryant CA, Mishra A, Maletic JI, Sharif B (2018) iTrace: eye tracking infrastructure for development environments. In: Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications, Association for Computing Machinery, New York, NY, USA, p 105, URL https://doi.org/10.1145/3204493.3208343

Guarnera DT, Behler JAC, Sharif B, Maletic JI (2025) Automated fixation error correction to support eye tracking studies on source code. Proc ACM Hum Comput Interact 9(3):1–17, DOI 10.1145/3725829, URL https://doi.org/10.1145/3725829

Hansen ME, Goldstone RL, Lumsdaine A (2013) What makes code hard to understand? CoRR abs/1304.5257:1–19, URL http://arxiv.org/abs/1304.5257, 1304.5257

Hollender N, Hofmann C, Deneke M, Schmitz B (2010) Integrating cognitive load theory and concepts of human–computer interaction. Computers in Human Behavior 26(6):1278–1288, DOI https://doi.org/10.1016/j.chb.2010.05.031, URL https://www.sciencedirect.com/science/article/pii/S0747563210001718, online Interactivity: Role of Technology in Behavior Change

Hunter JD (2007) Matplotlib: A 2d graphics environment. Computing in Science & Engineering 9(3):90–95, URL https://doi.org/10.1109/MCSE.2007.55

Just MA, Carpenter PA (1980) A theory of reading: from eye fixations to comprehension. Psychological review 87(4):329–354, URL https://doi.org/10.1037/0033-295X.87.4.329

Keshk AM, Dyer R (2023) Method chaining redux: An empirical study of method chaining in Java, Kotlin, and Python. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, IEEE Press, New York, NY, USA, MSR, pp 546–557, URL https://doi.org/10.1109/MSR59073.2023.00080

Kevic K, Walters BM, Shaffer TR, Sharif B, Shepherd DC, Fritz T (2015) Tracing software developers' eyes and interactions for change tasks. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, p 202–213, URL https://doi.org/10.1145/2786805.2786864

Kohn T, van Rossum G, Bucher II GB, Talin, Levkivskyi I (2020) Dynamic pattern matching with Python. In: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Association for Computing Machinery, New York, NY, USA, DLS 2020, p 85–98, URL https://doi.org/10.1145/3426422.3426983

Kuchling AM (2021) Functional Programming HOWTO. https://docs.python.org/3/howto/functional.html

Letovsky S (1987) Cognitive processes in program comprehension. Journal of Systems and Software 7(4):325–339, DOI https://doi.org/10.1016/0164-1212(87)90032-X, URL https://www.sciencedirect.com/science/article/pii/016412128790032X

McNemar Q (1947) Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika 12:153–157, URL https://doi.org/10.1007/BF02293996

Obaidellah U, Al Haek M, Cheng PCH (2018) A survey on the usage of eye-tracking in computer programming. ACM Comput Surv 51(1), URL https://doi.org/10.1145/3145904

de Oliveira B, Ribeiro M, da Costa JAS, Gheyi R, Amaral G, de Mello R, Oliveira A, Garcia A, Bonifácio R, Fonseca B (2020) Atoms of confusion: The eyes do not lie. In: Proceedings of the XXXIV Brazilian Symposium on Software Engineering, Association for Computing Machinery, New York, NY, USA, SBES '20, p 243–252, URL https://doi.org/10.1145/3422392.3422437

Park K, Weill-Tessier P, Brown N, Sharif B, Jensen N, Kölling M (2023) An eye tracking study assessing the impact of background styling in code editors on novice programmers' code understanding. In: 19th ACM Conference on International Computing Education Research (ICER), ACM, New York, NY, USA, p 444–463, URL https://doi.org/10.1145/3568813.3600133

Peitek N, Siegmund J, Apel S (2020) What drives the reading order of programmers? An eye tracking study. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020, ACM, New York, NY, USA, pp 342–353, URL https://doi.org/10.1145/3387904.3389279

Peng Y, Zhang Y, Hu M (2021) An empirical study for common language features used in Python projects. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, New York, NY, USA, pp 24–35, URL https://doi.org/10.1109/SANER50967.2021.00012

Pennington N (1987) Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology 19(3):295–341, DOI https://doi.org/10.1016/0010-0285(87)90007-7, URL https://www.sciencedirect.com/science/article/pii/0010028587900077

R Core Team (2018) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL https://www.R-project.org/

Rahman MR, Rahman A, Williams L (2019) Share, but be aware: Security smells in Python gists. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, New York, NY, USA, pp 536–540, URL https://doi.org/10.1109/ICSME.2019.00087

Rajan H, Nguyen TN, Dyer R, Nguyen HA (2021) Boa website. http://boa.cs.iastate.edu/boa/

Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in GitHub. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, FSE 2014, p 155–165, URL https://doi.org/10.1145/2635868.2635922

Rayner K (1998) Eye movements in reading and information processing: 20 years of research. Psychological bulletin 124(3):372–422, URL https://doi.org/10.1037/0033-2909.124.3.372

Rist RS (1986) Plans in programming: Definition, demonstration, and development. In: Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers, Ablex Publishing Corp., USA, p 28–47

Rodeghero P, McMillan C, McBurney PW, Bosch N, D'Mello S (2014) Improving automated source code summarization via an eye-tracking study of programmers. In: Proceedings of the 36th international conference on Software engineering, IEEE Press, New York, NY, USA, pp 390–401, URL https://doi.org/10.1145/2568225.2568247

Romanov V (2020) Evaluating importance of edge types when using graph neural network for predicting return types of Python functions. In: Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, Association for Computing Machinery, New York, NY, USA, SPLASH Companion 2020, p 25–27, URL https://doi.org/10.1145/3426430.3428135

Shapiro JR, Neuberg SL (2007) From stereotype threat to stereotype threats: Implications of a multi-threat framework for causes, moderators, mediators, consequences, and interventions. Personality and Social Psychology Review 11(2):107–130, URL https://doi.org/10.1177/1088868306294790

Sharafi Z, Shaffer T, Sharif B, Guéhéneuc YG (2015a) Eye-tracking metrics in software engineering. In: 2015 Asia-Pacific Software Engineering Conference (APSEC), IEEE, New York, NY, USA, pp 96–103, URL https://doi.org/10.1109/APSEC.2015.53

Sharafi Z, Soh Z, Guéhéneuc YG (2015b) A systematic literature review on the usage of eye-tracking in software engineering. Information and Software Technology 67:79–107, URL https://doi.org/10.1016/j.infsof.2015.06.008

Sharif B, Maletic JI (2010) An eye tracking study on camelcase and under_score identifier styles. In: 2010 IEEE 18th International Conference on Program Comprehension, IEEE, IEEE Press, New York, NY, USA, pp 196–205, URL https://doi.org/10.1109/ICPC.2010.41

Shrestha N, Botta C, Barik T, Parnin C (2020) Here we go again: Why is it difficult for developers to learn another programming language? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 691–701, URL https://doi.org/10.1145/3377811.3380352

Siegfried RM, Liporace D, Herbert-Berger KG (2019) What can the reid list of first programming languages teach us about teaching cs1? In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Association for Computing Machinery, New York, NY, USA, SIGCSE '19, p 1256–1257, URL https://doi.org/10.1145/3287324.3293830

Siegmund J, Peitek N, Parnin C, Apel S, Hofmeister J, Kästner C, Begel A, Bethmann A, Brechmann A (2017) Measuring neural efficiency of program comprehension. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2017, p 140–150, DOI 10.1145/3106237.3106268, URL https://doi.org/10.1145/3106237.3106268

Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. IEEE Transactions on Software Engineering SE-10(5):595–609, DOI 10.1109/TSE.1984.5010283

Spencer SJ, Steele CM, Quinn DM (1999) Stereotype threat and women's math performance. Journal of experimental social psychology 35(1):4–28, URL https://psycnet.apa.org/doi/10.1006/jesp.1998.1373

Steele CM, Aronson J (1995) Stereotype threat and the intellectual test performance of African Americans. Journal of personality and social psychology 69(5):797, URL https://psycnet.apa.org/doi/10.1037/0022-3514.69.5.797

Storey MD (2005) Theories, methods and tools in program comprehension: Past, present and future. In: 13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA, IEEE Computer Society, pp 181–191, DOI 10.1109/WPC.2005.38, URL https://doi.org/10.1109/WPC.2005.38

Storey MD (2006) Theories, tools and research methods in program comprehension: past, present and future. Softw Qual J 14(3):187–208, DOI 10.1007/S11219-006-9216-4, URL https://doi.org/10.1007/s11219-006-9216-4

Sweet DL (2020) nonpar: A Collection of Nonparametric Hypothesis Tests. Sweet, D. Lukke, URL https://CRAN.R-project.org/package=nonpar, r package version 1.0.2

the pandas development team (2020) pandas-dev/pandas: Pandas. URL https://doi.org/10.5281/zenodo.3509134

TIOBE Software BV (2023) TIOBE Index for July 2023. https://tiobe.com/tiobe-index/

Tukey JW (1949) Comparing individual means in the analysis of variance. Biometrics 5(2):99–114, URL https://doi.org/10.2307/3001913

Turner R, Falcone M, Sharif B, Lazar A (2014) An eye-tracking study assessing the comprehension of C++ and Python source code. In: Proceedings of the Symposium on Eye Tracking Research and Applications, Association for Computing Machinery, New York, NY, USA, ETRA '14, pp 231–234, URL https://doi.org/10.1145/2578153.2578218

Von Mayrhauser A, Vans A (1995) Program comprehension during software maintenance and evolution. Computer 28(8):44–55, DOI 10.1109/2.402076

Waskom ML (2021) seaborn: statistical data visualization. Journal of Open Source Software 6(60):3021, URL https://doi.org/10.21105/joss.03021

Wells MB, Kurtz BL (1989) Teaching multiple programming paradigms: A proposal for a paradigm general pseudocode. In: Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education, Association for Computing Machinery, New York, NY, USA, SIGCSE '89, p 246–251, URL https://doi.org/10.1145/65293.71222

Wiedenbeck S, Evans NJ (1986) Beacons in program comprehension. SIGCHI Bull 18(2):56–57, DOI 10.1145/15683.1044090, URL https://doi.org/10.1145/15683.1044090

Yang CK, Wacharamanotham C (2018) Alpscarf: Augmenting scarf plots for exploring temporal gaze patterns. In: Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, CHI EA '18, p 1–6, URL https://doi.org/10.1145/3170427.3188490

Yang Y, Milanova A, Hirzel M (2022) Complex Python features in the wild. In: Proceedings of the 19th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '22, p 282–293, URL https://doi.org/10.1145/3524842.3528467

Zhang Z, Xing Z, Xia X, Xu X, Zhu L (2022) Making Python code idiomatic by automatic refactoring non-idiomatic Python code with Pythonic idioms. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, p 696–708, URL https://doi.org/10.1145/3540250.3549143

Zhang Z, Xing Z, Xia X, Xu X, Zhu L, Lu Q (2023a) Faster or slower? Performance mystery of Python idioms unveiled with empirical evidence. In: 2023 IEEE/ACM 45th International Conference on Software Engineering, IEEE Press, New York, NY, USA, ICSE, pp 1495–1507, URL https://doi.org/10.1109/ICSE48619.2023.00130

Zhang Z, Xing Z, Xu X, Zhu L (2023b) RIdiom: Automatically refactoring non-idiomatic Python code with Pythonic idioms. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings, IEEE Press, New York, NY, USA, ICSE-Companion, pp 102–106, URL https://doi.org/10.1109/ICSE-Companion58688.2023.00034

Zyrianov V, Guarnera DT, Peterson CS, Sharif B, Maletic JI (2020) Automated recording and semantics-aware replaying of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, New York, NY, USA, pp 464–475, URL https://doi.org/10.1109/ICSME46990.2020.00051