University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

Fall 12-2-2022

An Empirical Study on the Classification of Python Language Features Using Eye-Tracking

Jigyasa Chauhan
University of Nebraska-Lincoln, jchauhan2@huskers.unl.edu

Follow this and additional works at: https://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Chauhan, Jigyasa, "An Empirical Study on the Classification of Python Language Features Using Eye-Tracking" (2022). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 230.
https://digitalcommons.unl.edu/computerscidiss/230

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

AN EMPIRICAL STUDY ON THE CLASSIFICATION OF PYTHON LANGUAGE FEATURES USING EYE-TRACKING

by

Jigyasa Chauhan

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Robert Dyer

Lincoln, Nebraska

December, 2022

AN EMPIRICAL STUDY ON THE CLASSIFICATION OF PYTHON LANGUAGE FEATURES USING EYE-TRACKING

Jigyasa Chauhan, M.S.

University of Nebraska, 2022

Adviser: Robert Dyer

Python, currently one of the most popular programming languages, is an object-oriented language that also provides language feature support for other programming paradigms, such as functional and procedural. It is not currently understood how support for multiple paradigms affects the ability of developers to comprehend that code. Understanding the predominant paradigm in code, and how developers classify the predominant paradigm, can benefit future research in program comprehension as the paradigm may factor into how people comprehend that code. Other researchers may want to look at how the paradigms in the code interact with various code smells. To investigate how developers classify the predominant paradigm in Python code, we performed an empirical study while utilizing an eye-tracker. The goal was to see if developers gaze at specific language features while classifying the predominant paradigm and debugging code samples. The study includes both qualitative and quantitative data from 29 Python developers, including their gaze fixations during the tasks. We observed that participants seem to confuse the functional and procedural paradigms, possibly due to confusing terminology used in Python, though they do gaze at specific language features. Overall, participants took more time classifying functional code. The predominant paradigm did not affect their ability to debug code, though they gave lower confidence ratings for functional code.

ACKNOWLEDGMENTS

I would like extend my sincere gratitude to my advisor Dr. Robert Dyer for his consistent and incessant guidance, support, and encouragement to help understand and explore the field of computer science. I am extremely thankful to him for letting me be a part of the ESQuaReD lab and accomplish this thesis satisfactorily. I would thankfully acknowledge my personal thanks to Dr. Bonita Sharif for her guidance and her relentless efforts to help to pursue my research. Also, I would like to thank Dr. Rahul Purandare for his valuable feedback on my research and taking the time to participate on my thesis committee. Lastly, I would thank the University of Nebraska–Lincoln for accepting me to be a part of there culture and providing resources for our research. My special thanks to my colleagues and lab mates for their support and encouragement. I would also like to thank my parents, friends and guardians for their trust and motivation throughout the journey.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Most modern programming languages have a predominant programming paradigm they support, where often that paradigm is object-oriented. Yet they are also multi-paradigm in the sense that they typically also support other programming paradigms, such as procedural or functional in addition to its predominant paradigm. One example of such a language is Python, which is one of the more popular and currently fastest growing programming languages [1]–[3]. To date, little is known about how developers utilize the different programming paradigms available to them.

An initial study in this direction [4] 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 then developed an automated classification script that runs in the Boa language [5], [6]. During the manual process, they noted that the human judges did not fully agree on what the predominant paradigm of some files were. It was not clear how humans even are judging such predominant paradigm. They also did not go beyond simply classifying the paradigm(s) observed in their dataset to see if the predominant paradigm of a Python script hinders developers comprehension or ability to debug that code.

Understanding how developers use different programming paradigms available to them and possible effects of those paradigms on their comprehension is an important topic. Such knowledge can help guide future researchers who are investigating code smells in Python [7], [8] as they could correlate paradigm-specific smells or see if new paradigm-specific smells are occurring and yet to be defined. It has been observed that the first language taught to students in early courses of computer science affects their comprehension of programming [9] and developing Python code in a predominant paradigm that is different than their first language could have an affect on developers. It is important for studies that are looking into dynamic frameworks using Python functions like lambda, for mapping [10]. Further studies, for example, researchers exploring pattern matching using functions, first-class objects along with pre-existing features [11], studies that are using Python packages for analyzing relationship structure using features definitions, imports and calls [12] demand the need to understand Python features among the developer community.

Further, Alexandru *et al.* [13] showed that developers have a loose understanding of terms and definitions related to Python. Also, Python and its features have been studied for a long time now [14] and the community has been trying to work on simplifying its paradigms for a long time. Further, when a developer transitions from an imperative paradigm to other and more recent ones like object-oriented and functional, it becomes harder for them to grasp new concepts associated with those features [15]. Python provides a variety of features with respect to each paradigm especially functional, object-oriented, and procedural.Therefore, it is of utmost importance for Python developers, and students to understand the Pythonic way of using Python.

These problems inspire us to look into how developers understand Python's paradigm-specific language features. In this study, we aim to first see if we can determine how developers classify the predominant paradigm in Python code, hoping to see if developers are aware of the predominant paradigm and what, if any, language-level features guide them to that knowledge. Then, we investigate if the choice of a predominant paradigm in Python affects the ability of developers to localize bugs in that code.

Eye-tracking has been evolving and is being extensively used within Software Engineering research. The methodology helps in capturing participants' visual attention and hence the gaze using certain metrics and practices [16]. There are published systematic literature reviews that help the research community to calculate and define the metrics used for eye-tracking [17]. The software engineering community has been understanding how developers gaze, debug, and read through programming languages. To understand developer behavior it is essential to understand the eye gaze and fixations. Studies have shown eye fixations are drivers of human visual cognition [18].

With this empirical study, we observe and investigate the classification and debugging of predominant paradigms in Python. We collect data using eye-tracking when the participant reads the source code. To collect data, we gathered 30 participants who had beginner to expert level of experience with Python. All the participants were trained on classifying Python features and then were asked to read source code and answer questions. After each category, the participant was interviewed. For the interviews, the participant was asked a set of questions and the audio was recorded. We then used these results to perform a mixed-method analysis. We use the analysis to understand the correlation between the confidence levels of the participant and the classification of the paradigm. The analysis also told us about how difficult was it for the participant to classify each Python paradigm (functional, object-oriented, procedural, and mixed). Our investigations show that paradigms are important for developers to understand and debug Python code.

In this thesis, we aim to answer the following research questions:

RQ1 How difficult is it for developers to classify the predominant Python paradigm?

RQ2 How accurately do developers classify the predominant paradigm in Python code?

RQ3 Do developers fixate their gaze on specific Python language features when classifying predominant paradigms?

RQ4 Does the predominant paradigm affect how long developer's take to debug logical errors?

RQ5 Does the predominant paradigm affect a developer's ability to debug logical errors?

After analyzing data from our participants, we found the following key results:

1. Developers spent longer classifying functional paradigm compared to other paradigms.
2. We observe that most of the developers were confused between classifying procedural and functional paradigms.
3. We found that, regardless of paradigm, developers gazed at specific language features when classifying.
4. Bug localization tasks had lower confidence levels compared to the classification tasks. Despite that, the accuracy was higher than the classification tasks.
5. We found no relation between paradigm and ability to debug. However, a few participants claimed that it is difficult to debug in the functional paradigm.

We investigate these research questions and provide data and evidence about developer behaviors while classifying predominant paradigm and localizing bugs in different paradigms. The contributions of this thesis are summarized as follows:

1. the first eye-tracking study to determine use of Python paradigms and features;
2. a study to observe developer's eye gaze while reading Python code in different paradigms;
3. analyzing how accurately do developers determine Python paradigms;
4. potential ways to read/debug a certain paradigm using the qualitative approach to predict task difficulty;
5. analysis on how developer's approach to classify changes when provided different code lengths with different Python paradigms; and
6. discussion on how collected data can help connect to various paradigms and promote better teaching and training for Python developers.

In the next chapter, we provide some background for this study and discuss some prior, related works. Then in Chapter 4 we provide the approach for our study. In Chapter 5 we discuss and analyze the results of the study. We then discuss some of the implications of those results in Chapter 6 and we conclude in Chapter 7.

Chapter 2

Related Work

In this chapter, we discuss previous studies investigating Python use.

2.1 Multi-language Studies

Eye movement studies are conducted in multiple ways to investigate the developers' behavior in particular activities such as reading code, seeking information on stack overflow, and polyglot programming. Peterson *et al.* [19] introduced an exploratory study on the information-seeking behavior of developers on stack overflow using an eye-tracking infrastructure. In this study, developer participants were assigned to the task which is creating human-readable summaries of methods and classes in large Java projects. With the eye-tracking data collected from the source codes and stack overflow documents while developers complete their tasks, the study found that developers inspect the text more than a title in a stack overflow. Code snippets were the next frequently observed element. Moreover, when developers switch between the stack overflow platform and the Eclipse Integrated Development Environment (IDE), they often checked method signatures and then switched to code and text elements on stack overflow. As they investigated that developers check the text more than a title in stack overflow, this study would like to investigate which Python features would get more programmers' attention when they read the Python source code.

Kochhar *et al.* [20] performed studies on various languages by building regression models to observe developer's behavior while bug fixing. There found most of the errors occur when the language is used in a multiple-language setting. The languages they used were C++, Java, and Objective C. There can be an extension to study and it might be interesting to observe similar results with respect to Python. It is interesting to observe how developers might think using multi-languages might simplify the complexity but it might be not the case for all languages.

Brilliant and Wiseman [21] studied three programming language paradigms (procedural, object-oriented, functional) and programming languages that feature those paradigms. They wanted to find an optimal first programming language for computer science students. They listed the feature, advantages, and disadvantages of each paradigm. Eventually, they concluded that there is no strong reason to pick any of those paradigms as the first programming paradigm for the students. This study was conducted in 1996, but now, with the aid of eye-tracking, we believe we can gather more information about how people actually interact with those paradigms and provide some insight into finding the optimal programming paradigm.

Uesbeck *et al.* [22] investigated polyglot programming through a randomized controlled trial conducted under the context of database programming. Since the study is about polyglot programming, coding tasks were given to participants and the tasks require multiple programming languages which are Java and one of three SQL-like embedded languages. This research found significant differences in how developers from different experience levels of coding use polyglot techniques. Their study finds that programmers with less experience tend to program faster with a hybrid language approach. Similarly, this study draws comparisons between paradigms and we observe find Object-Oriented the easiest of all paradigms since they are accurate and take lesser time to classify and comprehend.

Mayer and Bauer [23] analyzed the utilization of multiple programming languages for open-source projects. The authors discuss how many different languages are used per open source project and also provide information on how they categorize the language as domain-specific or general-purpose language. They also implemented regression analysis after getting an approximate number of co-occurrences. Their results show that mean of 5 general purpose languages are used in open source projects and 1/4þof the project is written in a domain-specific language.

Chakraborty *et al.* [24] discussed different programming languages on Q& A sites. They majorly focus on Go, Swift, and Rust over Stack Overflow. The authors implemented LDA and Stack Overflow responses for providing more resources to developers while they were working with a new language. In contrast to these studies we analyze Python language thoroughly by comparing different paradigms. Much similar to these studies our work focuses over understanding developers behavior while reading and debugging.

2.2 Code Summarizing

Abid *et al.* [25] observed developers' reading behavior while they summarize Java 5 methods. Unlike prior studies done by Rodeghero *et al.* [26] which only used short source codes in isolation, Abid's study allowed programmers to scroll through the source code and switch between the files to make a realistic working environment during the summarizing task. Their results indicate that they tend to focus on the method body more than the signature. The amount of gaze time and frequency of revisit were major components of the analysis and other results have been derived from them in Abid's study. One of the other results is that experts tend to write their summaries from source code lines that they read the most. Like this study, this study gives summarizing tasks to participants by enabling scrolling on the source code to see how they gaze at Python features during their task.

Abid *et al.* [25] studied how developers summarized methods specifically focusing on longer code methods that allowed scrolling and switching files.They draw comparisons between experts and novices and their approach to revisiting code methods or control flow. They observed the experts revisited more of the control flow statements. Although experts had major revisits on methods while they increased. Our study focused on majorly computer science students and we also found most of them used skimming as their approach and control flow.

Rodeghero *et al.* [26] investigated how java developers summarized code and wrote English summarized of methods given. They further utilize the observed techniques to build a novel automated code summarizing tool. The authors use eye-tracking and use VSM tf/IDF approach to understand the developer's gaze on keywords. Whereas we observe the developer's gaze using eye-tracking fixations and collecting audio during the interview and transcribing their responses. There study found that developer's used keywords from the source code and used them in their code summaries.

Peterson *et al.* [19] monitored how developers gazed at Stack Overflow. They observed developers looked at the titles and the code snippets when tasked to create human-readable code summaries focusing on code elements and token-wise granularity. Similarly, our study looks at the token-level fixations to understand the developer's reading behavior while they focus on keywords or Python features. In contrast, we look into Python features that have not been investigated before. Therefore, we think it would be interesting to observe the developer's behavior with Python language.

2.3 Code navigation behavior

Kevic *et al.* [27] provides us with the first-time study performed on open-source code to understand developer's behavior while navigating through the code for change tasks. The authors found out that the developer's gaze on small parts of methods on the data flow aspect of it. This study looks into how developers are focused on logic like the one that has the bug and most of them follow the approach of skimming, reading line by line, and controlling flow.

Busjahn *et al.* [28] conducted a study to compare eye movements between novices and expert programmers when they read and comprehend short natural language texts and Java programs. This research shows that novices read source code more linearly than experts, and also read the natural language more linearly than source code. As one of the novel contributions, this study designed local and global gaze-based measurements to characterize linearity in reading source code. This study adopts these measurements to characterize linearity in reading source code from top to bottom and from left to right. Similarly, our study focuses over developer's navigation behavior with respect to the approach. We look into how data flow, control flow, and skimming. On the contrary we found developers highly depend on skimming through the code to classify a predominant paradigm.

2.4 Python Paradigms

Floyd [15] talked about the impact of programming paradigms design. He mentions how there are different phases and defined two stages namely, the top-down design approach and the second phase followed is known as structured programming also called by the author as methods of the level of abstraction.

Wells and Kurtz [29] proposed a paradigm general pseudo-code that would help students learn different paradigms. The paper explicitly addresses difficulties in transitioning from imperative to other paradigms like object-oriented, functional, and logical problems. The technique provides a pseudo-code in the form of a super-set of paradigms which can be translated to any target language if needed by a developer who is new to a certain paradigm.

Bunkerd *et al.* [30] analyzed how the history of programming affects the code's naturalness due to the diverse backgrounds of coders. So they compare two main parameters consisting of the number of contributors to a project and the diversity of the project by its contributors. Their results came up with an idea of the relationship between naturalness and contributor's diversity. They observed less predictability in code with more contributors. Similarly, our study focuses on Python paradigms especially on human behavior shuffling between different paradigms like functional, object-oriented, and procedural. In contrast, we investigate how they classify paradigms based on Python features rather than addressing just the approach and reading behavior. We think it is important to study developer behavior on how certain features affect the paradigm judgment.

2.5 Python as a Programming Language

Shrestha *et al.* [14] inspected the gap for developers to learn a different language. They conducted an empirical study in which they went over Stack Overflow and looked for various buggy instances which could have originated due to a lack of programming language. Next, they also interview professionals asking a set of questions in which they relate their prior knowledge to a new programming language. They found most of the developers had faulty assumptions in relating to a new language. Within an interview experience, they also found that having prior knowledge may actually hamper their ability to learn a new language.

Peng *et al.* [31] investigated the common Python features in Python projects. To understand Python features better the authors propose a tool that would help the research community in better task building and make code less error-prone. The results talked about how developers chose testing and safety checks and avoided complex features that are more prone to errors. Further, they also found that gradual typing and keyword-only arguments have been used less frequently by developers. Further, they talk about how developers create their decorators and are concerned about `ImportError`.

Bafatakis *et al.* [32] investigates SO compliance issues with Python code. They studied the projects over Stack Overflow and talk about violations per a statement on Python code to understand code style compliance. The results show the top 10 violations that are caused during a certain style of code compliance. Most of the violations exist due to bad white spaces.

Shrestha *et al.* [14] implemented a tool known as Transfer Tutor to compare the learning curve between languages developer already knows and don't know with Python and R respectively. The tutorial can be a stepping stone for programmers who want to code in a different language and face difficulties adapting to another similar language.

Alexandru *et al.* [13] looked at different code architectures specifically into a Pythonic way of programming in Python. They observe how some terms have been loosely related to Python and other modern programming languages like Scala, Ruby over GitHub, and StackOverflow. They point out wrongly implemented and practiced terms in the Python ecosystem. Similarly, this study also points out audio snippets from developers that are confused and have a loose understanding of function definitions.

These previous works analyzed Python code from StackOverflow and focused on one language feature, however, our study pulled out existing code from GitHub and cover all the Python features necessary for understanding the paradigm classification. In similarity we focus on coding the Pythonic way and investigate by interviewing participants about their approach to debug a task.

In the next chapter, we provide background information about paradigms in Python and how to classify them.

Chapter 3

Background

In this chapter, we provide relevant background information about classification of predominant paradigm in Python.

Python, along with other modern languages, supports multiple programming paradigms. They can be classified into declarative, which includes functional and logic paradigms, and imperative programming, which includes object-oriented and procedural. Python is predominantly object-oriented but also supports other paradigms like procedural and functional. However, there have been discussions on what the best paradigm to use might be. For example, the Django framework [33] supports class-based and function-based views. Some articles claim function-based views are easier to read and understand [34], but the framework provides both and is moving more toward the class-based paradigm. Both of the approaches have a set of pros and cons and it is upon the programmer how to use it and when.

Here we briefly describe some of the features Python supports and how we can classify Python code according to those features. This is based largely on the work of Dyer and Chauhan [4]. We re-use the running example from their work and show it in Listing 3.1 and explain how they would classify this particular source 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 [35]. The full feature mapping is shown in Table 3.1.

Listing 3.1: Example Python code with each statement classified into paradigm(s)

```
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
```

If we consider the example code show in Listing 3.1, we can classify the overall code as object-oriented the majority 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. Whenever 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, and 10) as functional. But the `wrapper` function is being returned in the method, on line 6. This is an exam-

Table 3.1: Reference sheet for Python feature classification (from [4])

Python Language Feature	**Imperative**	**Procedural**	**Object-Oriented**	**Functional**
`if else elif`	x			
`while` loop	x			
`break`	x			
`continue`	x			
`assert`	x			
`del`	x			
array indexing	x			
`pass` (inside loop)	x			
`pass` (inside class)	x		x	
`pass` (inside def)	x	x		
`return`		x		
function (`def`)		x		
nested function (`def`)		x		
`class` declaration			x	
inheritance			x	
method (`def`)			x	
`with`	x		x	
`try`	x		x	
`except`	x		x	
`finally`	x		x	
`raise`	x		x	
`for` loop	x			x
(`not`) `in` operator	x			x
`yield`	x			x
function-as-arg				x
`lambda` functions				x
list comprehension				x
decorators				x
generator expressions				x
iterators (`__next/iter__()`)	x	x	x	x
`send()` (into generator)		x		x
`iter()`		x		x
`map()`		x		x
`sorted()`		x		x
`filter()`		x		x
`any()`		x		x
`all()`		x		x
`itertools.*()`		x		x
`functools.*()`		x		x
`enumerate()`		x		x
`zip()`		x		x

ple 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 to only mark things 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. In this study, we will 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.

We utilize the Boa script from Dyer and Chauhan [4] in our study to classify each file. Their script relies on the classifications from the table as well as some heuristics they discovered during their manual classification process (such as the fact we do classify line 2 as imperative, but most other lines are not). This script is used to classify every source file used in the sub-tasks of our study as a ground-truth.

In the next chapter, we provide a detailed outline of our study design, various tasks designed, the background of participants, and training involved during the process. Next section would also talk about in detail the approach used to perform qualitative and quantitative analysis.

Chapter 4

Experimental Study

This chapter discusses details of our approach, such as the task definitions, study procedures, and analysis methods.

4.1 Study Overview

We design our study in two parts: classification and bug localization. The classification tasks aim to provide support to answer the first three research questions, while the bug localization tasks provide support to answer the remaining two research questions. Participants complete four tasks for each part of the study–eight in total. First, for classification, participants are asked to classify the predominant paradigm (functional, procedural, or object-oriented) on some example code. Code is provided in three different lengths (small, medium, and large) to see if code length affects the classification task. Second, for bug localization, participants are given four codes (cube, factorial, largest, and palindrome) and asked to debug and identify a logical error in the code. Each of the four programs has four variants (functional, procedural, object-oriented, and mixed), where each variant has the same logical error inserted into it. During all tasks, we use eye tracking hardware to track the developer's gaze to see what feature(s) they are looking at. We then perform qualitative and quantitative analysis on the observations.

4.2 Task Categories and Questions

We divided study tasks into two categories, where each category had four tasks. Each task was allocated a maximum of 10 minutes for participants to complete. A full list of the code used for each task for both categories is provided in Appendix B.

Table 4.1: Overview of all tasks in this study

Task Categories	**Questions**	**Tasks**			
1. Classification	A. Functional	i. small	ii. medium	iii. large	
	B. Mixed	i. small	ii. medium	iii. large	
	C. Object-oriented	i. small	ii. medium	iii. large	
	D. Procedural	i. small	ii. medium	iii. large	
2. Bug Localization	A. Cube	i. OO	ii. procedural	iii. functional	iv. mixed
	B. Factorial	i. OO	ii. procedural	iii. functional	iv. mixed
	C. Largest	i. OO	ii. procedural	iii. functional	iv. mixed
	D. Palindrome	i. OO	ii. procedural	iii. functional	iv. mixed

In Table 4.1 we describe all the questions asked on the Google form while performing tasks for categories.

4.2.1 Classification Category

The first task category was designed to support answering RQ1–RQ3. These tasks are classification tasks, where participants are asked to classify the predominant paradigm of some code. We obtained real-world Python source file URLs using Boa's [5] "2021 Aug/Python" dataset. From that dataset, files were chosen in three different length categories. The lengths were determined using the number of statements in the file. There were three lengths:

- small: 10-15 statements
- medium: 16-30 statements

- large: 31-45 statements

There were a total of four tasks (procedural, object-oriented, functional, and mixed) and each task had the same questionnaire completed (see Appendix A.3 for the specific questions). Participants were asked to classify the code examples based on the predominant paradigm, as well as what percent of the code belonged to each specific paradigm.

An example of a Task-1 like task can be seen in Listing 3.1, where each statement is classified into paradigm(s). Each task used a different code example (see Appendix B.1 for all code listings). The code examples were shuffled across participants, but each participant was given one code example from each paradigm while also ensuring they saw code examples of all three lengths.

The code was displayed in the IDE with a font size of 18 points. Each subtask was timed with a maximum allowed time of 10 minutes to complete. An eye tracker started logging at the beginning of each task and stopped at the end, when participants submitted their questionnaire.

4.2.2 Bug Localization Category

The second task category was designed to support answering RQ4–RQ5. These tasks are bug localization tasks to measure how well participants comprehend the given code. Participants were asked to find and explain a bug in the given source code. There were four different algorithms provided:

1. cube of a number
2. largest number in a list
3. palindrome of an input string

4. factorial of a number

For each, a single logical bug was introduced. We had four variants of each code, one for each paradigm (see Appendix B.2 for all code listings). Each variant had the same logical bug applied to it. Each expected the same input and generated the same output. The code examples were shuffled across participants, but each participant was given one code example from each paradigm as well as one from each algorithm. After each task, participants were asked the same questionnaire (see Appendix A.4) where they were asked to explain what the bug was and the line it occurred on.

The code was displayed in the IDE with a font size of 18 points. Each task was timed with a maximum allowed time of 10 minutes to complete. An eye tracker started logging at the beginning of each task and stopped at the end, when participants submitted their questionnaire.

4.3 Participants and Grouping

The study included a total of 31 participants, including 2 that had to be excluded because of poor calibration (1) and no prior experience in Python (1). We collected and analyzed data for the remaining 29 participants. We compared similarities and differences between populations of the age group of 19 and older. All participants were students were from the University of Nebraska–Lincoln. The majority of the students were from a computer science background and others were from biological system engineering, statistics, computer engineering, and other similar fields. The participants were not provided any course incentives but were compensated with a $25 virtual Visa card.

Every participant was provided with guided training and a reference sheet. The entire study took 1.5 to 2 hours per participant. During the study, all participants filled out the pre-questionnaire at the start and post-questionnaire at the end, where they rated their level of expertise. Out of all participants, 23.3% rated themselves as novices, 66.7% as intermediates, and 10% as experts.

Each participant was provided with eight tasks in total, four in each category. We had two categories: classification and bug localization. For classification, each participant was given one code from each of the three paradigms and a mixed code. Each code was also different sizes. For each paradigm, we had three different sizes (small, medium, large). 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 we had four programs (cube, factorial, largest, and palindrome). For each program, we had four variants representing the four different paradigms. Tasks were assigned to ensure roughly equal coverage among each, so roughly one quarter of participants saw a functional version of Cube, one quarter saw an object-oriented version of Cube, one quarter saw a procedural version of Cube, and the remainder saw a mixed version of Cube. This was done for each of the four programs. Table 4.2 shows an example grouping for five participants.

4.4 Eye Tracking Apparatus and Environment

To track the participants eyes, we used a Tobii TX300 eye-tracker running in 60Hz frequency. Participants utilized Eclipse as the integrated development environment (IDE) for viewing all code snippets. An Eclipse plugin called iTrace [36] 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 [37]. The iTrace plugin provides an output of two XML files containing the full session logs. The files are further analyzed by running fixation filters, such as the Identification by Velocity Threshold (IVT) fixation filter [38] using default threshold numbers. Next, databases are generated using the fixation filter to generate a list of identified gazes, per task.

4.5 Study Variables

In this study we have some specific dependent and independent variables. The independent variables were the correct paradigm and correct bug detection. Some of the dependent variables were accuracy, confidence levels, and prediction. Another dependent variable is task duration. The measure of time completion per task gave us the total time they spent providing answers to the tasks.

4.6 Study Procedure

Participants were allocated up to two hours to complete entire study. After signing a consent form, participants filled out the pre-questionnaire. They were then given the training task. Next, each participant had to be calibrated with the eye tracker. Once the calibration was done, the participant was ready to perform tasks. The participants were asked to read code in Eclipse and answer the questionnaires provided as Google forms using an iPad. After each task group, an audio-recorded interview was performed. At the end of the study, the participants filled out the post-questionnaire.

4.6.1 Participant Training

In order to perform the study tasks, we first trained all participants. Also, they learned during training how to use properly sit to enable eye tracking. The moderator walked all participants through a reference sheet that consisted of mock code snippets for Task 1 (classification) and Task 2 (bug localization). Initially, the par-

Table 4.2: Assigning participants to specific tasks

	1. Classification Tasks			
Participant ID	**A. Functional**	**B. Mixed**	**C. Object-oriented**	**D. Procedural**
1	small	medium	large	small
2	medium	large	small	medium
3	large	small	medium	large
4	small	medium	large	small
5	medium	large	small	medium

	2. Bug Localization Tasks			
Participant ID	**A. Cube**	**B. Factorial**	**C. Largest**	**D. Palindrome**
1	func	oo	proc	mixed
2	oo	proc	mixed	func
3	proc	mixed	func	oob
4	mixed	func	oo	proc
5	func	oo	proc	mixed

ticipants were shown training code as shown in Listing 4.1 to explain how paradigms (functional, object-oriented, and procedural) using Python features are classified by each statement. They were also provided with a feature classification table from Dyer and Chauhan [4], as shown in Table 3.1.

Participants were then asked to classify the predominant paradigm of the code and could utilize the feature table. Here 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 `class MyNumbers`. While understanding every feature-wise classification per statement, they were also made aware of how one statement can belong to multiple paradigms. They then chose one predominant paradigm from the three paradigms: functional, procedural, and object-oriented. If code appeared to have more than one predominant paradigm, they could classify it as mixed. If someone failed the training, we remove their data from analysis.

For Task 2, participants were given the code example in Listing 4.2. This example has a bug on line 21, where the answer should be appended to the previous answer using `+=` instead of `-=`. Participants were then asked to find the line containing a bug and explain the fix.

Listing 4.1: Training code with paradigms classified (from [4])

```
class MyNumbers:         # func oo
    x = 1                #      oo      imp

    def m(self):         #      oo
        def m3():        #      oo
            return 1     #      oo proc
        y = m3()         #      oo proc
        return y         #      oo

    def __iter__(self):  # func oo
        self.x = 1       #      oo
            return self  #      oo

    def __next__(self):  # func oo
        y = self.x       #      oo
        self.x += 1      #      oo
        return y         #      oo

 x = MyNumbers()         #      oo
```

Listing 4.2: Training code for Task 2, with a bug on line 21

```
 1 class Solution:
 2    def intToRoman(self, num: int) -> str:

 3        dic = {
 4           1000 : 'M',
 5           900 : 'CM',
 6           500 : 'D',
 7           400 : 'CD',
 8           100 : 'C',
 9           90 : 'XC',
10           50 : 'L',
11           40 : 'XL',
             10 : 'X',
             9 : 'IX',
             5 : 'V',
             4 : 'IV',
             1 : 'I',
         }
         ans = ""

         for key in dic.keys():
             val = num // key
             ans -= val * dic[key]
             num -= val * key

         return ans
```

4.6.2 Quantitative Analysis

To perform the quantitative analysis we use the iTrace toolkit provided by the iTrace plugin [36]. Every time the eye-tracking was stopped during the study, we got two different XML files generated from Eclipse and iTrace itself. We use the iTrace XML file to get the x- and y-coordinates. Since the toolkit provides only a line-/column-level mapping, we had to write a query to map to tokens in the source code. The query outputs the token the participant gazed.

We also extract the questionnaire data from the Google forms by downloading the data as CSV files and post-processing with Python scripts. We then plot graphs and generate tables to answer each research question.

4.6.3 Qualitative Analysis

We designed questionnaires to reinforce the qualitative aspect of our study. We created a pre-questionnaire, a post-questionnaire, as well as questionnaires for each of the tasks. The pre-/post-questionnaires ask participants about their background and opinions, and the task questionnaires ask participants about their perceptions toward the code snippets and the particular task. We use this information to aid our analysis and identify potential threats to validity. Additionally, we also validate our results and observations through our recorded audio interviews at the end of each task group.

For the audio interviews, we perform open coding on the individual responses to attach concepts to them. Open coding has been used previously in software engineering studies [39]. We include all text from the audio interviews, which we manually transcribed. Every participant answered a total of 8-10 interview questions for both tasks 1 and 2 categories. Begel and Zimmermann [39] describe the process divided into three parts: preparation, execution, and analysis.

Preparation The initial phase for the interview was designing a set of questions for the participants for each category. The questions were asked in the same order for all participants, allowing for some variation depending upon their prior responses already covering the question. Further, each response, along with its respective question, was printed on small index cards as shown in Figure 4.1.

Execution The next phase was the classification of cards into similar categories with the same questions. We had 11 different piles of card categories belonging to each research question and each pile was labeled with a description.

Analysis The last phase of the process was to come up with themes from different piles and merging piles with similar themes. The author analyzed and classified 315 cards and created 15 different themes. The results are discussed in the next chapter.

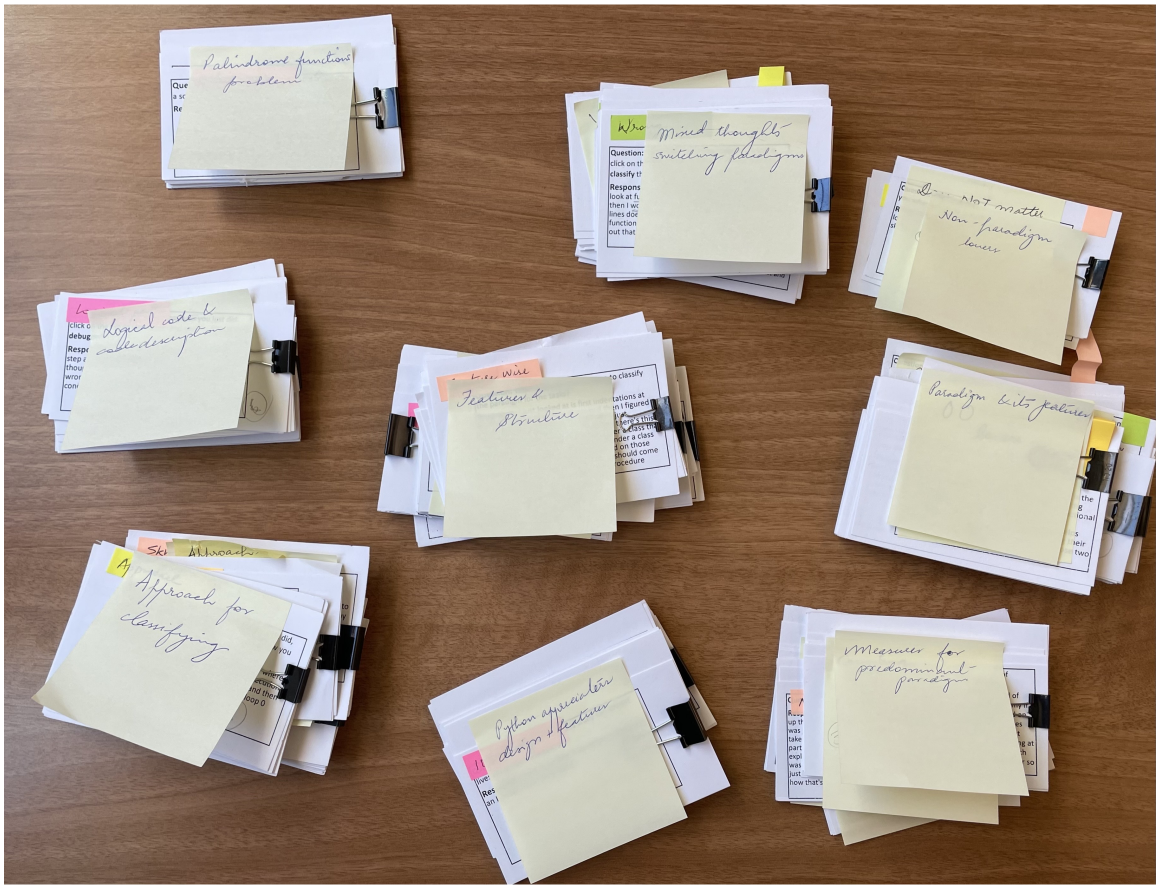
Palindrome functions problem
Mixed thoughts switching paradigms
Does NOT matter
Non-paradigm issues
Logical code & code description
Features & Structure
Paradigm bits features
Approach for classifying
Python association design features
Measures for predominant paradigm

Figure 4.1: Themes after card sorting

4.6.4 Qualitative data collected

We also designed questionnaires to reinforce the qualitative aspect of our study. We created a pre-questionnaire, a post-questionnaire, as well as questionnaires for each of the four tasks. The pre-/post-questionnaire ask the participant about their background and opinions, and the task questionnaires ask participant about their perception towards the code snippets. We use this information to aid our analysis and identify potential threats to validity.

We provide Table 4.3 in order to help readers understand how each data source was used to answer the research questions.

We mainly describe all the data from pre- and post-questionnaires and Tasks as

Task 1 questionnaire or Task 2 questionnaire, data from iTrace and eye-tracking as **eye-tracking data**, and audio interviews as **audio**.

Table 4.3: Methods used for each research question

	Data Source Used
RQ1	Task 1 questionnaire + eye-tracking data
RQ2	Task 1 questionnaire
RQ3	eye-tracking data
RQ4	Task 2 questionnaire + eye-tracking data
RQ5	Task 2 questionnaire

In the next chapter, we provide the results of our study.

Chapter 5

Results

This chapter discusses the quantitative and qualitative results of the study. First, we look at the demographics of the 29 participants.

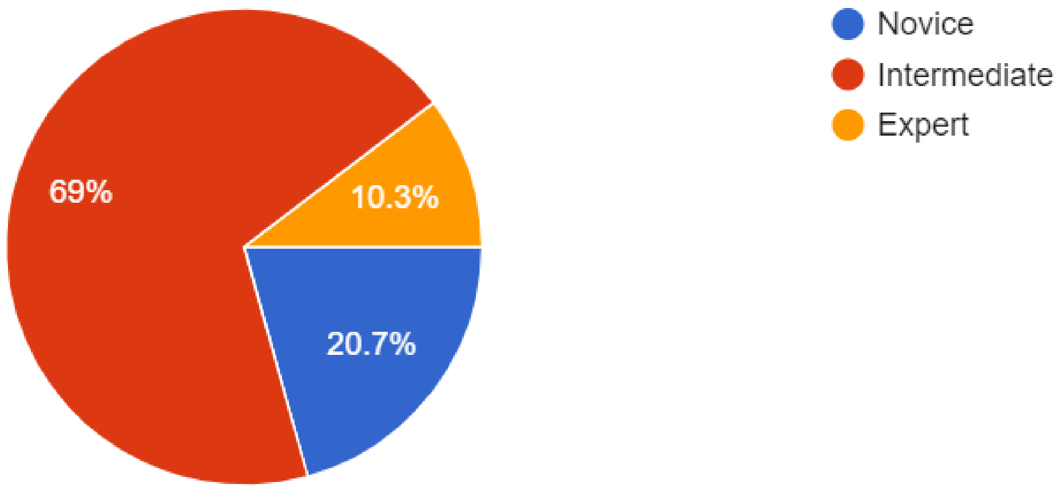

Figure 5.1: Survey: Self-reported rating for programming skills in Python

Figure 5.1 shows the ratio of our participants across the study can be split into different levels of self-reported experience with programming in Python. We asked this question in the post-questionnaire to avoid priming participants. We observe that 10.3% indicated they were experts, 69% intermediate, and 20.7% beginners.

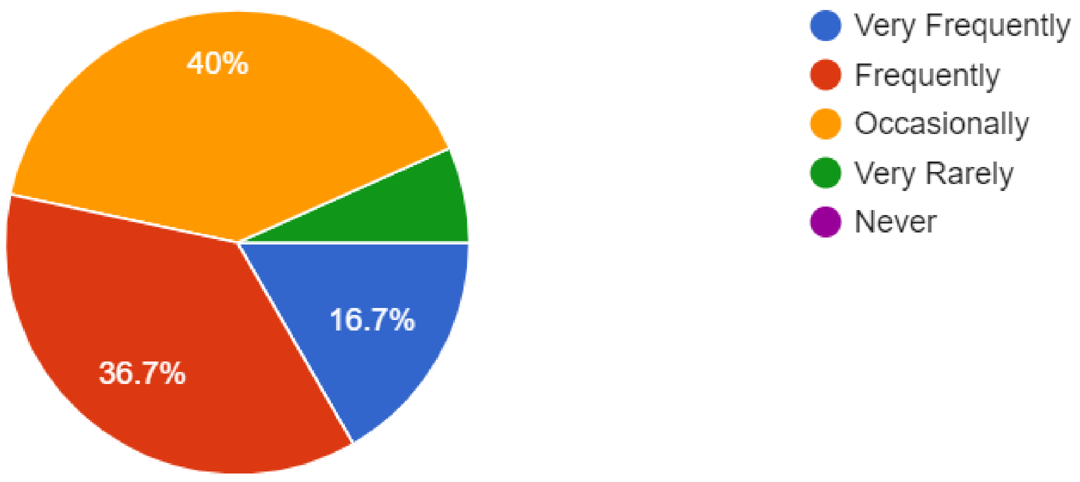

Figure 5.2: Survey: How often do you code using Python?

We also asked participants how often they code using the Python language. Figure 5.2 shows more than 42% of the participants code in Python (very) frequently, 40% code occasionally, and 6.6% rarely code in Python. We find there are no participants who never code in Python, so all participants included in our results have Python experience.

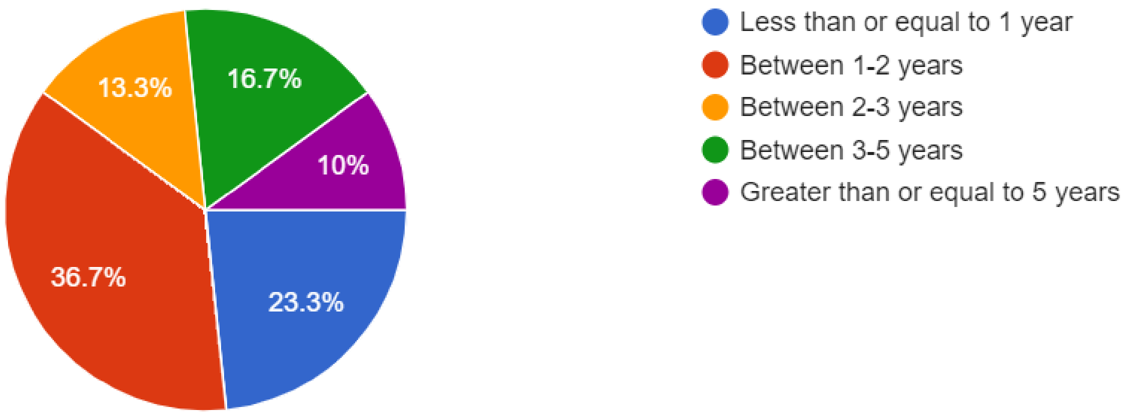

Figure 5.3: Survey: How long have you been coding using Python?

Figure 5.3 shows how long the participants have been coding in Python. Almost one quarter of the participants have up to one year of experience. 60% have up to two years experience. Only 10% of all participants have more than and equal to five years, which matches up with the number self-reporting as expert.

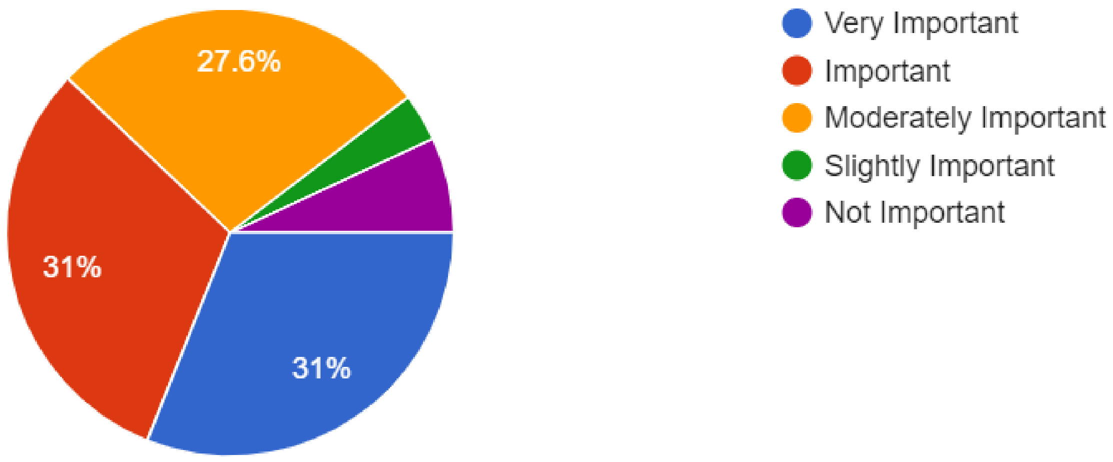

Figure 5.4: Survey: How important is it to code for you in Python?

We also wanted to see how participants view coding in Python. Figure 5.4 shows their responses to how important they think it is to code in Python. Almost two-thirds of participants think it is (very) important to code in Python. Only around 10% indicated it was slightly or not important. This shows the participants value the language we chose to study.

5.1 Quantitative Results

In this section, we discuss results specific to every task in the two different task categories.

5.1.1 Task 1: Classification

5.1.1.1 RQ1: How difficult is it for developers to classify the predominant Python paradigm?

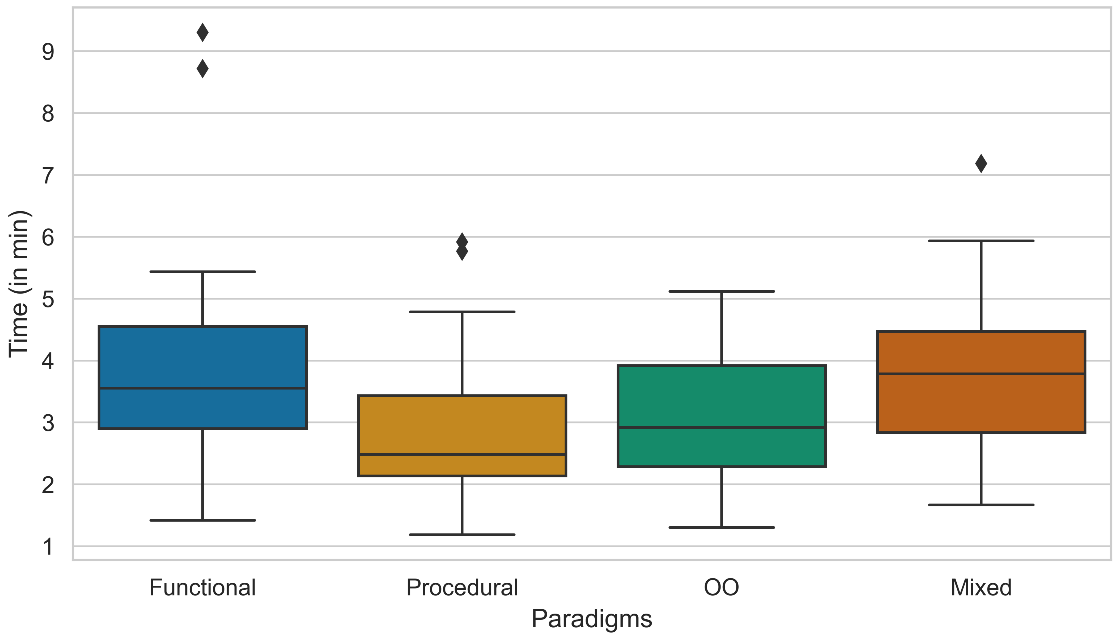

Figure 5.5: Task 1: Time taken for classifying different paradigms

Figure 5.5 describes the time taken by participants for each task. The time is calculated by the duration provided by the iTrace toolkit which is the time taken to look at the code and answer simultaneously. The participants were asked to notify the moderator when they were done looking at the code and the eye-tracker was stopped.

The graph shows the similar time taken in terms of functional and Mixed and the least time taken for procedural tasks. However, we also observe some outliers which were mostly participants who took longer to classify the task. Especially with the functional paradigm we observed took longer than the average time.

Further, we take a look in depth at the classification time change with respect to different lengths of the code snippet. We plot a graph for each different length (small, medium, and large) and the time taken in minutes.

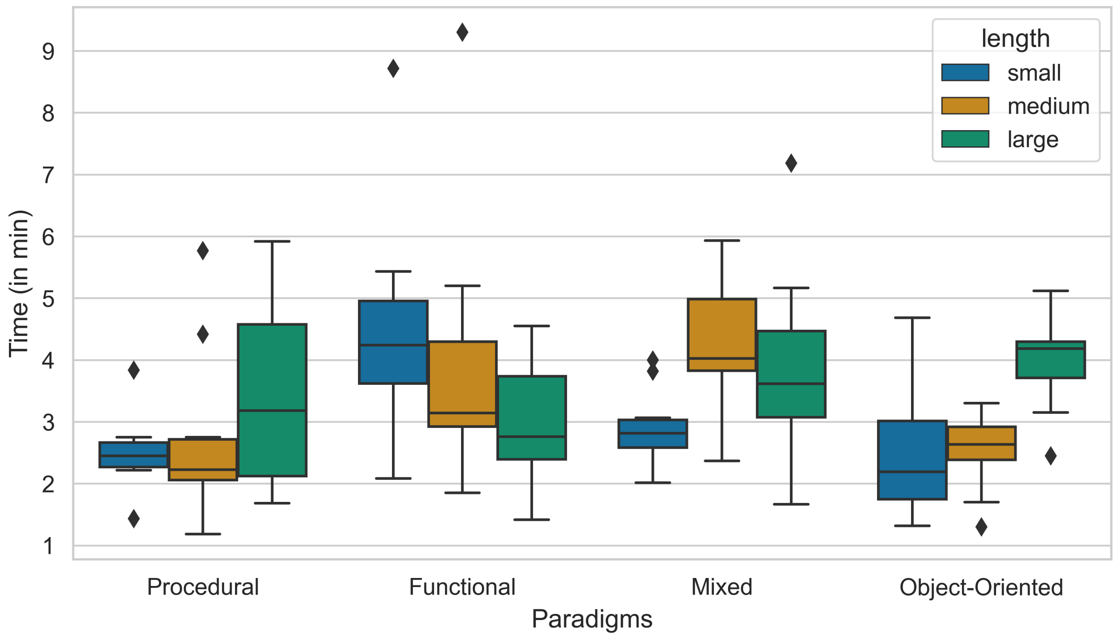

Figure 5.6: Task 1: Time taken w.r.t different lengths

The box-plot in Figure 5.6 shows the time taken with different lengths of various paradigms (small, medium, and large). We observe participants took longer for procedural large files and took minimum time for the small and medium size of files. For functional, the time taken decreases with the size increase of the file. Mixed shows us a variable result with the medium being the highest with respect to the time taken by the participant. The last paradigm, Object-Oriented took a long time when the length was large and comparatively less time if the length was small or medium.

Next, in Figure 5.7 we see how confident were the participants when they marked their overall confidence.

We plot a pie-chart to show the overall confidence for Classification category. More than 80% participant throughout the task. However, we do observe participants being Moderately, Slightly, and Not Confident while classifying files into various paradigms.

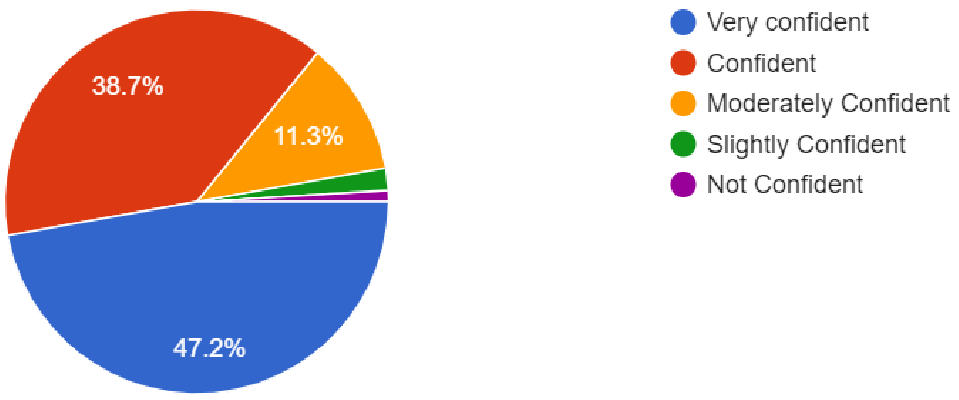

Figure 5.7: Task 1: Overall confidence (in percent)

In the next research question we discuss about how accurate was developer's judgment. To measure accuracy we consider two metrics - correct judgment and confidence level provided as Task1 Form. Then we plot the confidence level with respect to correct judgments.

5.1.1.2 RQ2: How accurately do developers classify the predominant paradigm in Python code?

Classification into various percentages

Table 5.1 shows the number of judgements made by participants as different percentages for each task which was procedural, object-oriented, functional, and mixed. We use the data provided by survey from Task 1. We observe 24 out of 29 participants classified the functional paradigm correctly. 25 participants out of 29 classified object-oriented paradigm correctly, and 16 out of 29 participants classified procedural paradigm correctly. We find a higher number in correct judgement of paradigms for functional and object-oriented. Less than half of the participants marked procedural paradigm incorrectly.

Table 5.1: Task 1: What do you consider the predominant programming paradigm for the code you read?

	Judgements			
Tasks	**Functional**	**Object-Oriented**	**Procedural**	**p-value**
Functional (small)	**9**	0	1	0.021
Functional (medium)	**5**	1	4	
Functional (large)	**8**	0	1	
OO (small)	2	**8**	0	0.108
OO (medium)	0	**7**	2	
OO (large)	0	**10**	0	
Procedural (small)	3	0	**7**	0.0001
Procedural (medium)	**5**	0	**5**	
Procedural (large)	2	3	**4**	
Mixed (small)	3	2	**4**	
Mixed (medium)	1	1	**8**	
Mixed (large)	0	**6**	4	

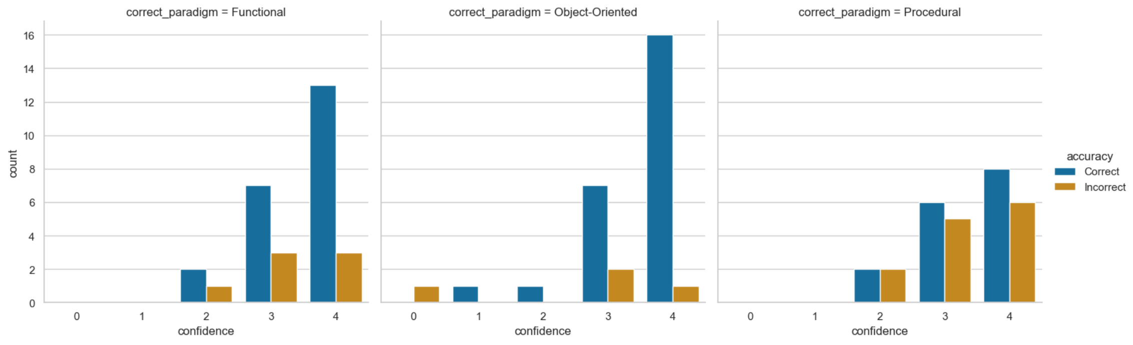

Figure 5.8: Task 1: Judgment vs confidence level

In Table 5.8 we observe that most of the participants were confident with the judgment of paradigms like functional and Object-Oriented. Whereas they were significantly confused with procedural. Half of the participants made an incorrect judgment about procedural paradigm. The interesting observation is that their confidence levels were equally high as their responses with correct responses for other paradigms.

This was interesting as it indicates that the participants were less aware of procedural and functional features, therefore they classified them wrong. To validate these results we also pick audios from the interview stack and understand that the participants were confused and had a hard time classifying `def():` and other procedural/functional features.

The box plot above shows how the classification of paradigms varied across other paradigms and across different lengths. We see some patterns for paradigms like functional, that show that the participants took lesser time as the length of the functional task was increased. We see a reverse pattern in terms of object-oriented as the time increases as the length of the task increases.

Along with one correct judgment for a predominant, we wanted to observe what do developers classify the code if it has more than one predominant paradigm. They were given the option to choose different paradigms and classify in into various percentages. The developer's got four choices to classify into less than 25%, 25–50%, 50–75%, and more than 75%.

Table specific to more than one predominant paradigm code

Table 5.2: Task1: What percentage of the code falls under the following paradigms?

	Procedural				**Functional**				**Object-Oriented**			
	≤ 25	$25-30$	$50-75$	≥ 75	≤ 25	$25-30$	$50-75$	≥ 75	≤ 25	$25-30$	$50-75$	≥ 75
Functional	8	10	3	8	16	12	1	0	3	3	4	**18**
Object-Oriented	8	9	10	2	2	3	3	**21**	12	9	7	1
Procedural	2	6	5	**15**	18	6	3	2	6	9	4	9
Mixed	7	5	3	**14**	4	8	11	6	10	11	5	3

We observe in Table 5.2 that participants were confident on their judgments for functional and object-oriented paradigms as they classified most of them as more than 75%. We found a split between multiple paradigms which is true in the case of mixed paradigm.

Table 5.3 shows how the participants understand 'mixed' as a paradigm. They classified most of the tasks as mixed out of the choices for mixed - yes, no, and maybe.

Further, we strengthen these results by providing Pearson's correlation Table 5.4. This correlation explains how two paradigms are dependent.

Table 5.3: Task 1: Do you think the code has more than one predominant paradigm? For example: functional, object-oriented, mixed, procedural

Tasks	Maybe	Yes	No
Functional (small)	1	3	**6**
Functional (medium)	2	**4**	**4**
Functional (large)	0	**5**	**5**
OO (small)	1	**6**	4
OO (medium)	1	**5**	4
OO (large)	0	**8**	2
Procedural (small)	0	**7**	3
Procedural (medium)	2	3	**5**
Procedural (large)	2	**7**	1
Mixed (small)	1	**7**	2
Mixed (medium)	2	**7**	1
Mixed (large)	2	**8**	0

Table 5.4: Pearson correlation between different paradigms

Statistical Analysis	Functional and Object-Oriented	Functional and Procedural	Object-Oriented and Procedural
Pearson	statistic=-0.668, pvalue=0.534	statistic=0.249, pvalue=0.839	statistic=-0.887, pvalue=0.305

We also performed SpearmanR and Kendalltau correlations Table 5.5. In this relation we observe the monotonic relation between variables. In this aspect our variables are the three paradigms (functional, object-oriented, and procedural). We see a stronger relation between functional and object-oriented.

Table 5.5: SpearmanR and Kendalltau correlations between different paradigms

Statistical Analysis	Functional and Object-Oriented	Functional and Procedural	Object-Oriented and Procedural
SpearmanR	correlation=-0.866, pvalue=0.333	correlation=0.5, pvalue=0.666	correlation=-0.866, pvalue=0.333
Kendalltau	correlation=-0.816, pvalue=0.220	correlation=0.333, pvalue=1.0	correlation=-0.816, pvalue=0.220

Table 5.6: Task 1: Approach with paradigm

Approach	Functional	Procedural	Object-Oriented	Mixed
Skim	27	25	28	23
Read line by line	10	11	14	18
Control flow	11	14	13	10
Data flow	5	2	3	4
Trace and execute	4	6	2	3
Trial and Error	0	0	0	1

Table 5.6 shows that developers are highly dependent on skimming for classification across all paradigms and task lengths.

5.1.1.3 RQ3: Do developers fixate their gaze on specific Python language features when classifying predominant paradigms?

We observed how developer's gaze at specific features with respect to every paradigm. Figure 5.9, Figure 5.11, Figure 5.12, and Figure 5.10 are the graphs that are specific to each paradigm. functional, mixed, object-oriented, and procedural.

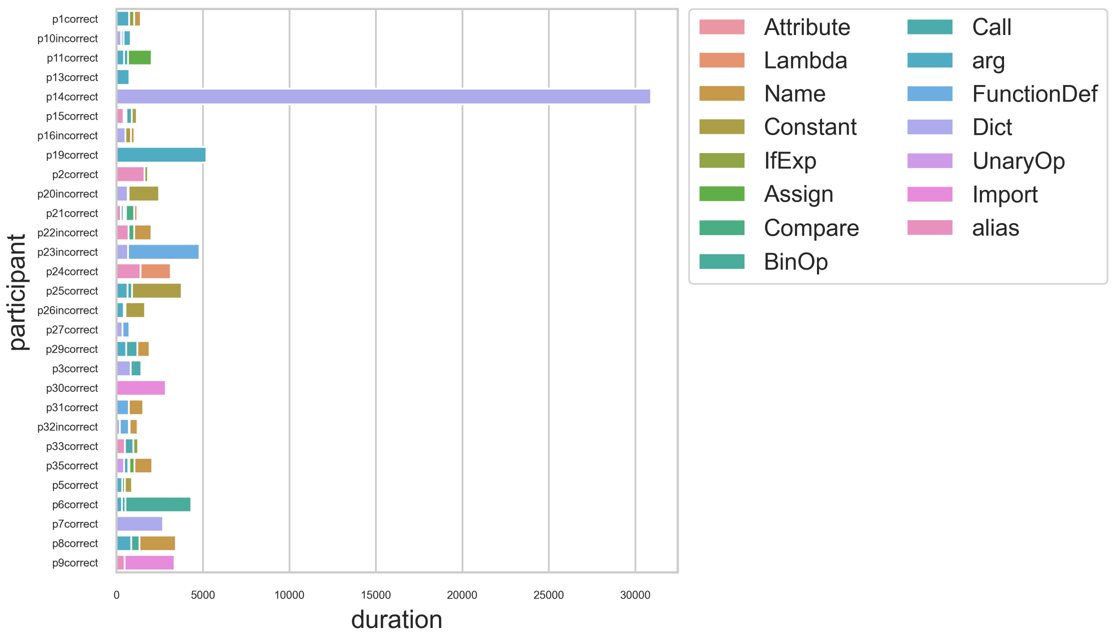

Figure 5.9: Eye-tracking data: Participant gaze for functional paradigm code snippets

We observe that participants look at certain features to classify language features with respect to paradigm.

Listing 5.1: Participant gaze on **Functional** code

```
import numpy as np
tansig = lambda n: 2 / (1 + np.exp(-2 * n)) - 1
sigmoid = lambda n: 1 / (1 + np.exp(-n))
```

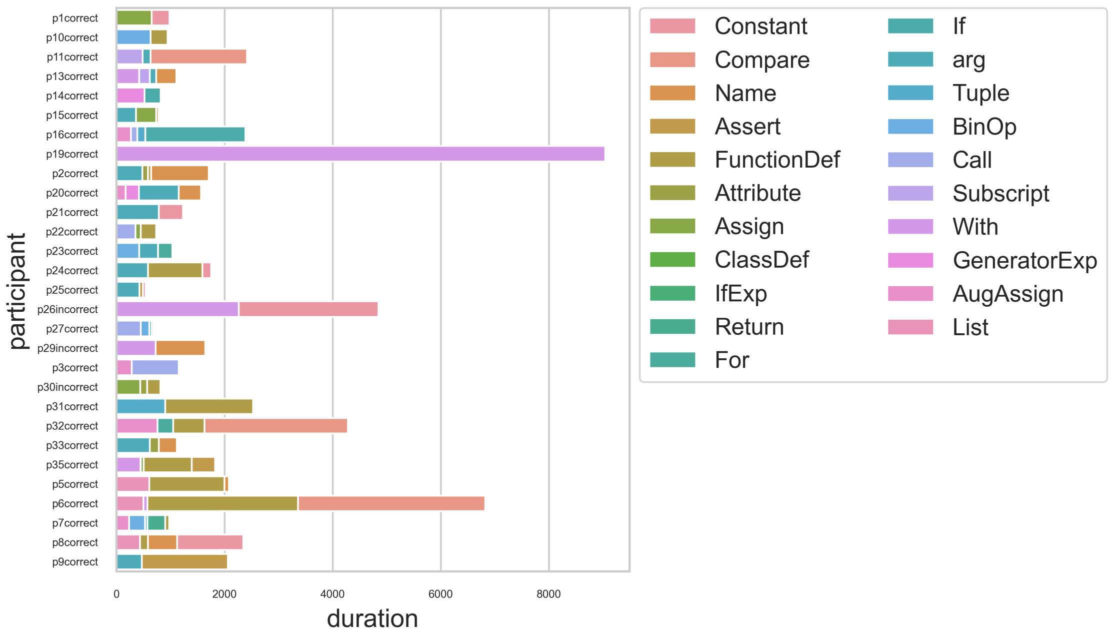

Figure 5.10: Eye-tracking data: Participant gaze for Mixed code snippets

```
hardlim = lambda n: 1 if n >= 0 else 0
purelin = lambda n: n
relu = lambda n: np.fmax(0, n)
square_error = lambda x, y: np.sum(0.5 * (x - y)**2)
sig_prime = lambda z: sigmoid(z) * (1 - sigmoid(z))
relu_prime = lambda z: relu(z) * (1 - relu(z))
softmax = lambda n: np.exp(n)/np.sum(np.exp(n))
softmax_prime = lambda n: softmax(n) * (1 - softmax(n))
cross_entropy = lambda x, y: -np.dot(x, np.log(y))
```

Further, we observe that each task with a specific paradigm and length for classification category, has been performed by four participants. Therefore, we manually highlight different fixations Listing 5.1 in for functional small task. We observe four different color gazes in red, blue, green, and orange. We see all participants gazed at `lambdas` and functions, and classified the paradigm as functional.

Listing 5.2: Participant gaze on **Mixed** code

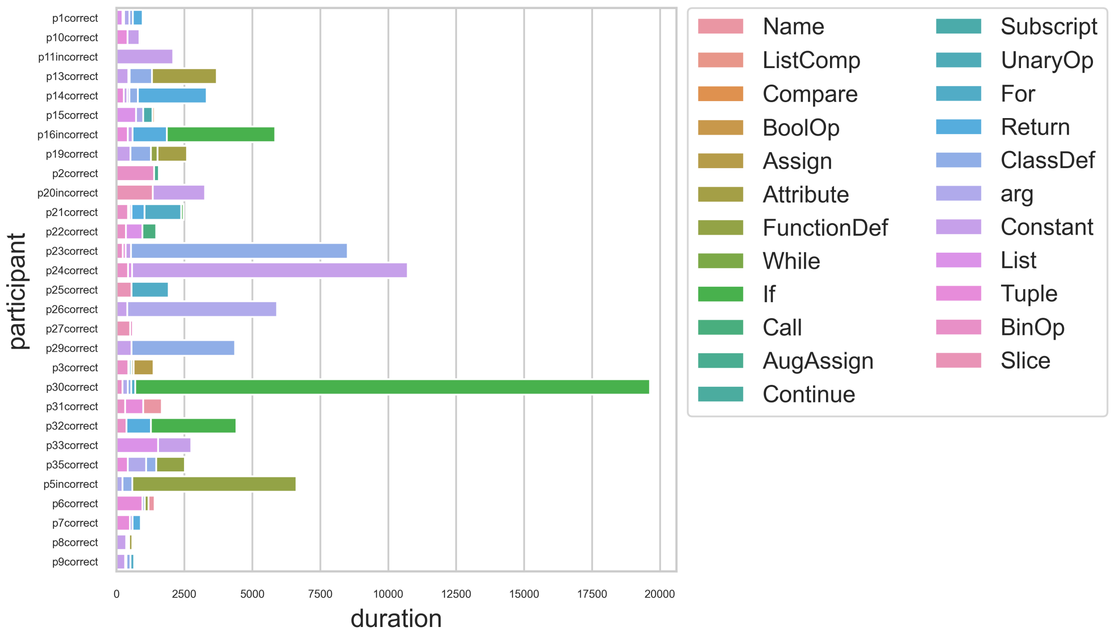

Figure 5.11: Eye-tracking data: Participant gaze for Object-Oriented code snippets

```
def test_personal_sendTransaction(accounts, rpc_client, password_account, account_password):
    initial_balance = rpc_client('eth_getBalance', [accounts[1]])

    with pytest.raises(AssertionError):
        rpc_client('personal_signAndSendTransaction', [{
            'from': password_account,
            'to': accounts[1],
            'value': 1234,
        }, "incorrect-password"])
    assert rpc_client('eth_getBalance', [accounts[1]]) == initial_balance

    rpc_client('personal_signAndSendTransaction', [{
        'from': password_account,
        'to': accounts[1],
        'value': 1234,
```

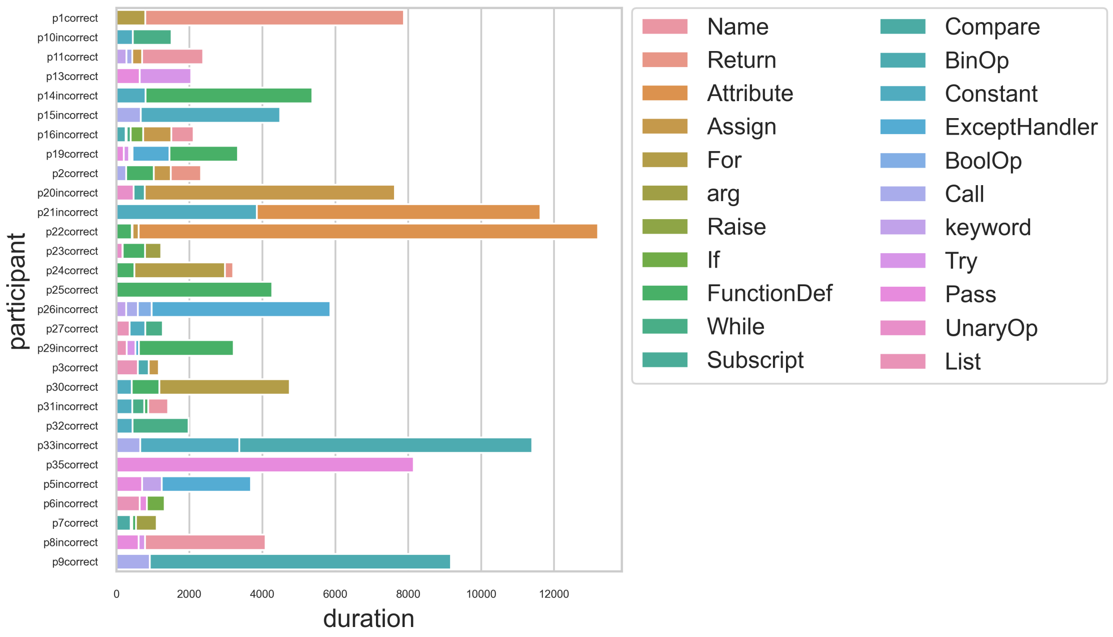

Figure 5.12: Eye-tracking data: Participant gaze for Procedural code snippets

```
    }, account_password])
    after_balance = rpc_client('eth_getBalance', [accounts[1]])

    assert after_balance - initial_balance == 1234
```

Similarly, among others one of the interesting results were from the mixed paradigms Listing 5.2. We saw the participants gaze at different features across all paradigms.

5.1.2 Task 2: Bug localization

For Task 2, we try and understand how long does the developer take to debug a specific file. We also observe if paradigms affect the ability to debug a developer. Further research questions look into how accurate was the error debugging as per a specific paradigm. This is checking the judgement on the bug line number and fixing the bug.

5.1.2.1 RQ4: Does the predominant paradigm affect how long developer's take to debug logical errors?

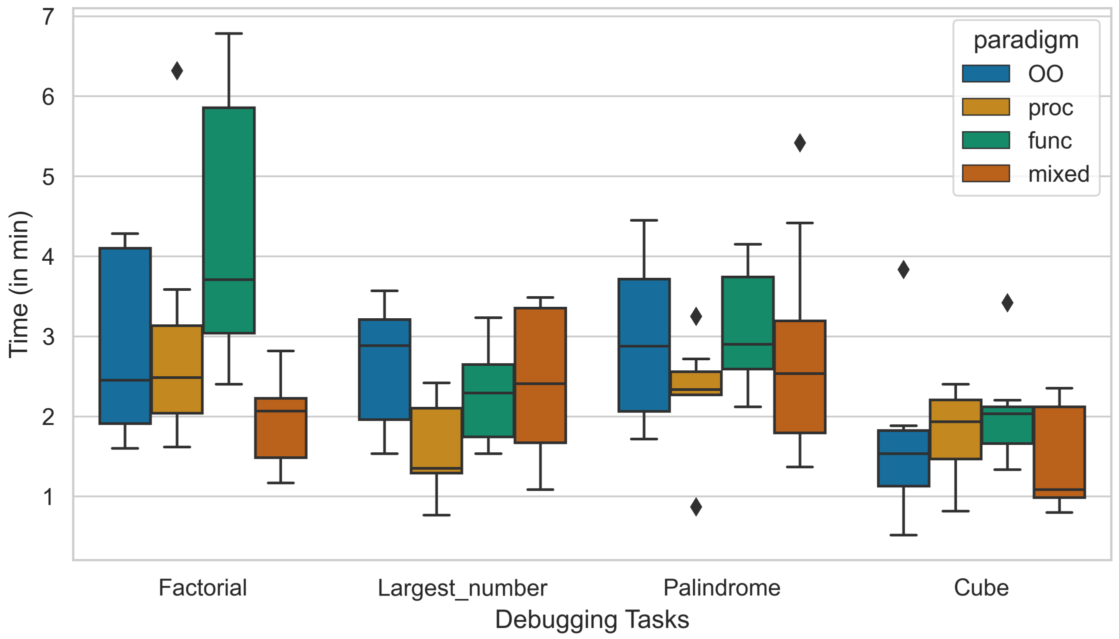

Figure 5.13: Task 2: Time taken to complete debugging

In Figure 5.13, we observe the time taken for participants with respect to different paradigms and tasks. There are four different tasks on the ordinate axis (factorial, largest number, palindrome, and cube). We see that the factorial task was difficult for most of the participants, especially when written in a functional paradigm. On the contrary, the cube was fairly easier than the other 3 tasks and hence participants took lesser time than other tasks.

In Table 5.7, we tabulate the results obtained from Figure 5.13 and include the length of the file. In this case the length of the file is the number of statements. We find that factorial functional takes the maximum time of 28 minutes with 10 statements. In comparison cube mixed took the least time of 10 minutes with 5 statements.

Table 5.7: Debugging with respect to task, paradigm and length

task	duration	length	paradigm
cube	13	11	oo
cube	14	8	proc
cube	14	4	func
cube	10	5	mixed
factorial	20	12	oo
factorial	20	7	proc
factorial	**28**	10	func
factorial	12	8	mixed
largest	14	14	oo
largest	11	11	proc
largest	15	6	func
largest	16	11	mixed
palindrome	21	11	oo
palindrome	**13**	8	proc
palindrome	21	6	func
palindrome	20	10	mixed

5.1.2.2 RQ5: Does the predominant paradigm affect a developer's ability to debug logical errors?

Table 5.8: Task 2: Correctness with paradigm

Paradigm	**Correct**	**Incorrect**
Object-Oriented	21	8
Procedural	27	2
Functional	23	6
Mixed	19	10

Table 5.8 shows us that participants debugged all the tasks with around 85% of correctness except mixed. We found no impact of paradigm on bug localization by the participants. Next, we observe their accuracy on basis of task difficulty.

In Table 5.9 we calculate and check the correctness for Task 2 bug localization by the participants. We observe that the participants were able to debug the cube, largest, and palindrome code more correctly than the factorial code.

Table 5.9: Task 2: Correctness with task

Task	Correct	Incorrect
Cube	25	4
Factorial	18	11
Largest	22	7
Palindrome	25	4

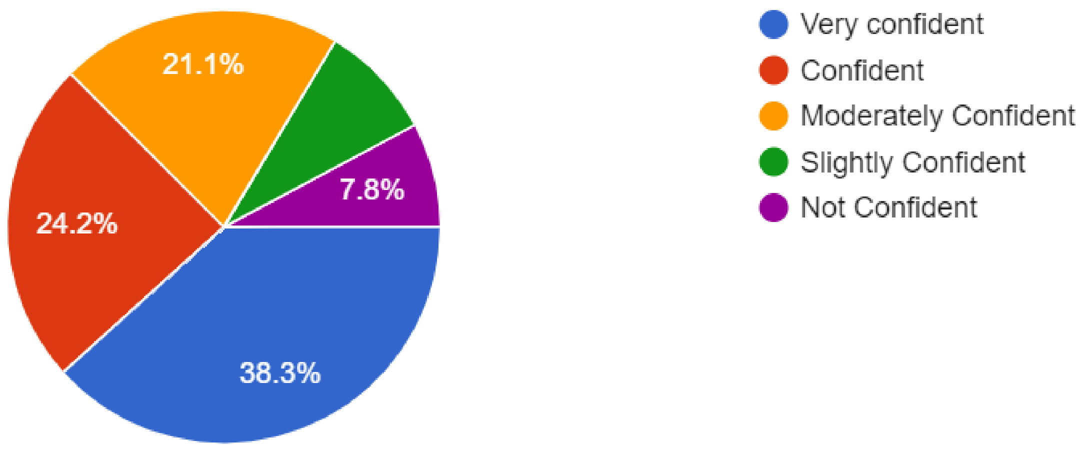

Figure 5.14: Task 2: Reported confidence levels

Another metric that we use to check the accuracy is the level of confidence of the participants. We find that more than 50% of the participants were confident in the bug localization task. However, we find these confidence levels to be lower than the classification category, where the participants were more than 85% confident.

In the next section, we discuss the limitations and possible threats to our study and how we mitigate them.

5.2 Qualitative Analysis

To better understand the view and thought process of the participant on how they classified and debugged the code, we performed two interview sessions. The interview was conducted after each task category. Once the data was collected, initially, the question and the response were transcribed and printed on flashcards. Next, we perform card sorting as per different questions. In total, we got 315 cards. We categorized them into 11 different slots based on 11 different questions.

Table 5.10 are the themes that came up during the process of card sorting. We filtered the data into 9 categories. Each category had sub-categories ranging from 2 to 7.

Table 5.10: Audio: Different Card Sorting Categories

Category Title	Cards	Sub-Categories
Approach	54	6
Importance of features	52	7
Code Logic and Description	14	3
Unawareness of functions	15	2
Switching on paradigms	32	5
Supporting Python paradigms	22	3
Unaffected by Python paradigms	5	2
Measure for predominant paradigm	22	4
Paradigm specific classification	57	6

From Table 5.10 we see that most of the programmers were very specific about the approach they used to read the code in order to perform either classification or bug localization. It is interesting to see a significant number of participants going back and forth to classify a predominant paradigm. Especially to classify procedural paradigms and functional. It is noted that most of the participants who marked a Procedural wrong tend to classify it as functional and vice-versa. `def` has been one of the major features, that confused the participants between procedural and functional paradigms. Below are a couple of code snippets that show audio responses from a couple of participants who were confused.

Approach The participants in this category talked about the approach they followed to classify and debug the task. Different approaches like skimming through the code, reading line by line, and observing control flow, and data flow were discussed and used to classify the tasks. Most of the participants were inclined towards skimming through the code regardless of how lengthy was the code. *"Typically, it's a quick skim to the code to see that it stands out and then going back in line by line depending on how familiar I was with how the code was written because of my experience level I would try to run the code in my head and thought we either how the information was passed or just the execution of individual functions"*

Importance of Features The participants in this category talked about the importance of Python features. The responses majorly talked about what particular features did they use to classify and debug the tasks. For example, the responses included def, lambda, for loops, and other features. *"I believe I did classify procedural for this task palindrome, I skimmed and looked here how Python is nicely tabbed in and out. everything was housed in a definition Initial skim and then for loops that there is nothing inside and something happening multiple times. After skim so I saw def is a function so procedural. But also, for loops which are more functional, so I saw a lot of functional going on inside also."*

Code Logic and Description The participants in this category talked about the flow and how they used the logic they followed to classify and debug the task. With questions related to how they debugged the file, the participants walked us through the entire control flow and point out the bug. *"I actually like read what the task is supposed to do and then identify the main part like here it's going to find the largest number so this is the Max function that does it and then here at clearly like identify that this sign should be reversed so sometime it just like experience I would say maybe like a lot of times that's why"*

Unawareness of functions The participants in this category talked about how the participants were confused and curious about to use of some features. One such example was debugging a palindrome task for a bug and the participants were not aware of the lower() function. *"I will have to do it again, but def I think I chose to be functional and then it was with and has assert."*

Switching on paradigms The participants in this category were extremely confused and most of them had incorrect judgments regarding the classification of the code. They started explaining paradigm A and they ended with re-considering their judgments of Paradigm B. *"so here this is the function definition and end it goes up to the end so I consider this entire thing to be a part of procedural OK and then I have this class definition here and these are the class methods so class is like within this procedural thing so I consider two predominant things here procedural and object oriented."*

Supporting Python paradigms The participants in this category appreciated Python and its ability to support various paradigms. For example, using loops helped them debug the task for the cube. These participants appreciated how developers can use object-oriented for data analysis and procedural if they need methods. *"I didn't know the classification myself and I do a lot of Python programming but this official formal classification I was never familiar with given the study I'll immediately go home, and you know try to practice on these things and check where my knowledge exactly" "I think it's important because everything doesn't come under object oriented and everything doesn't come under procedural simple program can be written in a procedural and complex and if you are writing a complex code in a simple terms you use object oriented program so I think it's important to do define this paradigms."*

Unaffected by Python paradigms The participants in this category discussed they do not pay attention to paradigms while they code in Python. Most of the participants in this category talked about using Python as needed for the task. They did not have a preference to choose over a paradigm. *"I do so much so simple work I don't normally think about more than just a simple one through procedural and functional kind of basis so I don't really get to ease object oriented programming to its fullest"*

Measure for predominant paradigm The participants in this category talked about what methods did they use to measure the predominant paradigm. Methods like counting statements, estimations, calculating percentages, and skimming through the code to check for more features specific to the paradigm. *"I didn't do any like math right I kind of just to the ballpark feel of like how many I saw in this kind of estimated."*

Paradigm specific classification The participants in this category talked about looking for certain features to judge a specific paradigm. *"I would say this is my functional to me because although the inside definition that is which is procedural but since it is covered in wrapped up inside a function so that's why I would say that this is functional right."*

5.3 Threats to Validity

In this section we discuss potential threats to the validity of our study. We classify these possible threats into three different categories: construct, internal, and external validity.

5.3.1 Construct Validity

Tasks were given to each participant in the same order, and this can cause unintended ordering effects. We partially mitigated this by randomizing the paradigm orders.

Procedures like task start and stop time might have been less accurate since it was done manually and we relied on the participants to inform the moderator when they were finished with each task.

5.3.2 Internal Validity

Given the fact that Python has so many features and each programmer might only use a subset of them, their reading pattern can be different, and a sample size of 29 might not be enough. In addition, all of our subjects were university students, so this might not be the best sample set to represent the Python developers community, i.e. our study might make some faulty generalizations.

Another threat to the validity is the unaddressed linearity. For example, in Python source code, some features might occur together and thus receive more attention from the programmer. For example, a functional code example that had a lot of lambdas on almost every line might have been very simple for some developers. To mitigate this threat we randomly picked real-world code files from Boa's 2019 Python dataset.

5.3.3 External Validity

In addition to the first two threats, since we are trying to understand the latent thought process of the subjects, there is a possibility of misinterpretation. The thought process is hard to capture, and though eye-tracking, and questionnaires we can accurately understand. Maybe the interpretation we derived is not how the programmer comprehends the code.

For Task 1, we chose tasks from a curated data set for Python. To mitigate the threat that dataset does not generalize to the Python population, we randomly pick the code based on three different lengths. For Task 2, we manually selected and wrote the code for the problems. This might lead to biases, since all code was written by a small set of authors. To help mitigate this threat, we ensure that same type of logical bug is present across all different paradigm for each task problem so the only variance is the paradigm used.

In the next chapter, we discuss some of the implications of our results.

Chapter 6

Discussion

In this chapter we discuss some possible implications of the results of our study.

The study showed that participants tend to take a similar time to classify all paradigms in Python, at a high (85%) confidence level regardless of paradigm. This seems to indicate that developers are good at finding paradigm-specific language features in Python. Peng *et al.* [31] found that developers chose certain Python features over others when writing Python code. Our study shows they are aware of specific features, so may make sense to see if their knowledge and their preferred choice of features correlate at all.

Our findings also show that the developers took longer to classify as the size of the file increases in the case of procedural, mixed, and object-oriented paradigms. However we see a different trend in the case of the functional paradigm, we found that developers took longer time to classify as the size of the file increased. This might have been because the functional paradigm is easier to identify as functional features, such as lambdas, look sufficiently different than non-functional features to make identifying them faster and easier. Therefore, with the increase in the size of the file, if they saw more lambdas highlighted in blue, they skimmed through the file and classified faster.

In detail, we also find developers classified a small mixed paradigm code faster, whereas they took longer to classify a large object-oriented paradigm. Another aspect to keep in mind is that these responses are not based on their judgments, but just on the total time, they took to classify the file from a specific paradigm. When we look into their judgments with respect to object-oriented we find that 86% of participants classified it accurately. If we take a deeper look at the large object-oriented paradigm, we might get a correlation in which as the size of the file increased from medium to large it significantly increases the time to classify, because of more statements of object-oriented code. We also find interview responses to validate our discussion regarding object-oriented paradigm. One participant said, *"I just skimmed the code I just looked over the blue highlighted paradigms class def return I looked at those keywords found out in the list classified."*

The study shows some interesting results on how human behavior classifies a paradigm as mixed when compared to the automated machine classification. Table 5.3 shows participants chose most of the tasks as having more than one predominant paradigm, except for medium-procedural and small-functional. These results show that most participants classified more than one paradigm as mixed, regardless of the specific percentages they classify each paradigm into. Here is one interesting percentage split a participant talks about - *"if I would choose predominant one then I would generally select 75% and for the others, I would look around and if I felt t that there were enough of them to define like a mixed one then I would set it somewhere between 50 to 75 and the other"*. However, the thresholds to classify as mixed are not clear, and indeed probably vary across developer. This does show that developers are aware of the fact Python is multi-paradigm and thus most Python code is, to varying degrees, mixed. What is not clear is how developers may have to change their mental model when switching between paradigms in the code, or if, as Peng *et al.* [31] showed, when they prefer specific features does that make them switch their mental model if the surrounding code has a paradigm that differs from their preferences.

Table 5.1 shows us results about how developers classify different paradigms and we see an interesting relation between procedural and functional paradigm classification. We observe that the participants were specifically confused about medium functional and medium procedural paradigms. As we observe, under functional out of 10 participants that performed that task, half of them classify it as functional but out of the other half, four classified it as procedural. It is interesting to see a similar trend under the procedural paradigm the participants classify half of it as procedural and another half as functional. There definitely seems to be confusion among developers about what procedural and functional mean in Python, most likely due to the terminology used by the language. This result shows there may be a need to better train new developers and clarify what these terminologies mean, especially in relation to specific paradigms. This might explain why Alexandru *et al.* [13] found that developers have a lack of understanding of Python definitions as they loosely term them over GitHub and StackOverflow.

Future studies can investigate in depth about there awareness and accuracy of features with respect to coding and simple natural text. We find most of our participants were specifically confused by the `def()` method. `def()` is a method in Python but often used as `def()` function. This confusion might be a result of the difference between the word function and functional. We find our one of participant explaining that "...def() function I think I chose to be functional ...". We might infer that as human there can be some confusion because of the word function implied to belong to a functional paradigm. Another interesting aspect of our results is the confidence level of the participant with respect to correct and incorrect judgments. We observe that confidence levels increase with correct judgments across all paradigms. On the contrary, confidence still rises with incorrect judgments. This result is significantly high with the procedural paradigm. These findings are interesting and we think the confidence levels per task might have increased due to performing similar tasks in sequence since procedural was not the first task in order to perform by any participant. There might be some effect of an increase in confidence with task performance in general. But it would be a great avenue to conduct a human study using procedural as a paradigm across languages and observe the developer's confidence and accuracy.

Further, the card sorting category helped us come up with different themes we see a maximum number of cards under the theme of approach. We found that skimming through the code was the top approach used by participants to classify the predominant paradigm. These results are similar to what we see in [28]. The participants also followed reading line by line, control flow, data flow, and trace and execute across all different paradigms. We show that there was no correlation between the self-identified approach by participants and the paradigm. With our findings tool, designers can use a particular way of code highlighting so that while skimming through new code, the developer can get the most about the code.

We notice that participants use the approach of reading line by line higher in mixed in comparison to other paradigms. One of the reasons this might be true is because a mixed paradigm has a combination of two or more paradigms, hence the participant considers looking at multiple features from different paradigms as we saw before. We also validate these results since reading line by line took longer time and this might be a reason why medium and large mixed paradigms took longer to classify.

For the bug localization category, we looked into the effect of paradigm on the developer's accuracy to debug a logical error. We also investigated the developer's debugging ability with the effect of different paradigms. We found no correlation between paradigm and bug localization for the largest, palindrome, and cube tasks. However, we see that the participants take a longer time to debug functional factorial tasks. This makes us turn towards the interview data and one of the participants talks about how they like to code the functional paradigm but find it harder to debug, *“The functional paradigms are harder for me to understand when I’m debugging because functional paradigms quickly change objects so fast”* . This makes us question that is there a trend with functional paradigms being hard to code. Functional paradigm has been significantly used lesser than procedural and object-oriented [4].

In the next chapter, we conclude and summarize our findings of this study.

Chapter 7

Conclusion

Despite the large and growing popularity of the Python language, little has been studied about how the language supports multiple paradigms and what affect those multiple paradigms might have on the ability to isolate bugs in the language. In this study, we used eye-tracking to first see if particular language features are used by developers 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 a functional paradigm and have lower confidence of that classification, compared to other paradigms. They also incorrectly classified procedural and functional code and found themselves confused between those two paradigms. Next, we observed that the developers gaze at paradigm-specific language features while classifying, regardless of the paradigm. We also found that predominant paradigm has no relation with the ability to isolate bugs, though participants rated their confidence in the task as lower. However, participants stated that code written in a functional paradigm was more difficult to debug.

In the future, we hope to investigate why participants found it harder to debug the functional code. We also want to investigate the boundaries where developers stop seeing the code as having a predominant paradigm and start viewing it as more mixed, so we can better understand when their mindset might change while reading or debugging mixed code. We also hope to survey some developers to get a better feeling for why specifically they seem to confuse procedural and functional paradigms in Python.

Appendix A

Questionnaires

In this chapter, we list all of the questionnaires or audio interview questions asked of each participant.

A.1 Pre-Questionnaire

The pre-questionnaire was given to each participant at the start of the study. The questions included:

1. What is the highest degree you have completed?
2. How often do you program in Python?
3. Which of the following programming languages can you read and comprehend?
4. How many years of experience do you have programming in Python?

A.2 Post-Questionnaire

The post-questionnaire was given to each participant at the end of the study. The questions included:

1. List any difficulties you had performing this study, if any.

2. How important is it for you to code in a specific programming paradigm?
3. How would you rate your programming in Python?
4. How was your overall experience during the study?
5. Please list any other comments that you have.

A.3 Task 1: Sub-Task Post Questionnaire

After each Task 1 sub-task, participants were asked these questions:

1. What is the most predominant paradigm for this code?
2. What is your confidence level for above classification?
3. Do you think the code has more than one predominant paradigm?
4. What are the various percentages for specific paradigm?
5. Check mark all various high-level constructs you see in the code above.

A.4 Task 2: Sub-Task Post Questionnaire

After each Task 2 sub-task, participants were asked these questions:

1. What is the paradigm of the code?
2. Do you think there is more than one predominant paradigm?
3. What line number is the bug on?
4. Explain how would you fix the bug?
5. What is your confidence level for your solution above?

6. Mention any additional comments for task-2.

A.5 Interview Questions

In this section, we list all interview questions asked at the end of each task group.

A.5.1 Task 1 Interview: Classification

1. What approach did you follow in order to classify the programming paradigms?
2. Did your approach change with the change in length of the file?
3. Pick a task and navigate us how did you classify this task?
4. How do you define predominant paradigm?
5. Which keywords did you gaze the most?

A.5.2 Task 2 Interview: Bug Localization

1. Pick a task and navigate us how did you debug this task?
2. What approach did you use to debug this file?
3. How do you think study can impact lives of Python developers?
4. Do you think it's important for you to code in different paradigms?
5. Were you forced to provide a solution?

Appendix B

Task Code Listings

In this appendix, we list all code listings used for every sub-task in the study.

B.1 Task 1 Code Listings: Classification

In this section we list all sub-task code for the classification Task 1. The code is real-world code from GitHub, as identified by our Boa query.

B.1.1 Functional Files

B.1.1.1 small length of code

Source: https://github.com/alexvlis/Shape-Recognition/blob/cc979394c4a2864c9a9b4a21b17e14880a6c6ae2/nnmath.py

```
import numpy as np
tansig = lambda n: 2 / (1 + np.exp(-2 * n)) - 1
sigmoid = lambda n: 1 / (1 + np.exp(-n))
hardlim = lambda n: 1 if n >= 0 else 0
purelin = lambda n: n
relu = lambda n: np.fmax(0, n)
square_error = lambda x, y: np.sum(0.5 * (x - y)**2)
sig_prime = lambda z: sigmoid(z) * (1 - sigmoid(z))
relu_prime = lambda z: relu(z) * (1 - relu(z))
softmax = lambda n: np.exp(n)/np.sum(np.exp(n))
```

```
softmax_prime = lambda n: softmax(n) * (1 - softmax(n))
cross_entropy = lambda x, y: -np.dot(x, np.log(y))
```

B.1.1.2 medium length of code

Source: https://github.com/laixintao/iredis/blob/4d93a1fb7736052e894ad54aeb4c2c4ef4fb5614/tests/unittests/command_parse/test_set_parse.py

```
def test_sadd(judge_command):
    judge_command(
        "SADD foo m1 m2 m3", {"command": "SADD", "key": "foo", "members": "m1↩
 m2 m3"}
    )
    judge_command("SADD foo m1", {"command": "SADD", "key": "foo", "members":↩
 "m1"})
    judge_command("SADD foo", None)


def test_sdiffstore(judge_command):
    judge_command(
        "SDIFFSTORE foo m1 m2 m3",
        {"command": "SDIFFSTORE", "destination": "foo", "keys": "m1 m2 m3"},
    )
    judge_command(
        "SDIFFSTORE foo m1",
        {"command": "SDIFFSTORE", "destination": "foo", "keys": "m1"},
    )
    judge_command("SDIFFSTORE foo", None)


def test_is_member(judge_command):
    judge_command("SISMEMBER foo m1 m2 m3", None)
    judge_command(
        "SISMEMBER foo m1", {"command": "SISMEMBER", "key": "foo", "member": ↩
"m1"}
```

```
    )
    judge_command("SISMEMBER foo", None)


def test_smove(judge_command):
    judge_command(
        "SMOVE foo bar m2",
        {"command": "SMOVE", "key": "foo", "newkey": "bar", "member": "m2"},
    )
    judge_command("SMOVE foo m1", None)
    judge_command("SMOVE foo", None)


def test_spop(judge_command):
    judge_command("SPOP set", {"command": "SPOP", "key": "set"})
    judge_command("SPOP set 3", {"command": "SPOP", "key": "set", "count": "3"})
```

B.1.1.3 large length of code

Source: `https://github.com/frankhjwx/osu-storyboard-engine/blob/78cf4e928f342398b08948e0f61ff4a04a38f8bc/Storyboard%20Engine/tools/easingFuncs.py`

```
Reverse = lambda func, value: 1 - func(1 - value)
ToInOut = lambda func, value: 0.5 * (func(2 * value) if value < 0.5 else 2 - func(2 - 2 * value))
Linear = lambda x: x
QuadIn = lambda x: x**x
QuadOut = lambda x: Reverse(QuadIn, x)
QuadInOut = lambda x: ToInOut(QuadIn, x)
CubicIn = lambda x: x**3
CubicOut = lambda x: Reverse(CubicIn, x)
CubicInOut = lambda x: ToInOut(CubicIn, x)
QuartIn = lambda x: x**4
QuartOut = lambda x: Reverse(QuartIn, x)
```

```python
QuartInOut = lambda x: ToInOut(QuartIn, x)
QuintIn = lambda x: x**5
QuintOut = lambda x: Reverse(QuintIn, x)
QuintInOut = lambda x: ToInOut(QuintIn, x)

SineIn = lambda x: 1 - math.cos(x * math.pi / 2)
SineOut = lambda x: Reverse(SineIn, x)
SineInOut = lambda x: ToInOut(SineIn, x)

ExpoIn = lambda x: math.pow(2, 10 * (x - 1))
ExpoOut = lambda x: Reverse(ExpoIn, x)
ExpoInOut = lambda x: ToInOut(ExpoIn, x)

CircIn = lambda x: 1 - math.sqrt(1 - x * x)
CircOut = lambda x: Reverse(CircIn, x)
CircInOut = lambda x: ToInOut(CircIn, x)

BackIn = lambda x: x * x * ((1.70158 + 1) * x - 1.70158)
BackOut = lambda x: Reverse(BackIn, x)
BackInOut = lambda x: ToInOut(lambda y: y * y * ((1.70158 * 1.525 + 1) * y - 1.70158 * 1.525), x)

BounceOut = lambda x: 7.5625 * x**2 if x < 1 / 2.75 else 7.5625 * (x - (1.5 / 2.75))**2 + 0.75 if x < 2 / 2.75 \
    else 7.5625 * (x - (2.25 / 2.75))**2 + 0.9375 if x < 2.5 / 2.75 else 7.5625 * (x - (2.625 / 2.75))**2 + 0.984375
BounceIn = lambda x: Reverse(BounceOut, x)
BounceInOut = lambda x: ToInOut(BounceIn, x)

ElasticOut = lambda x: math.pow(2, -10 * x) * math.sin((x * 0.075) * (2 * math.pi) / 0.3) + 1
ElasticIn = lambda x: Reverse(ElasticOut, x)
ElasticOutHalf = lambda x: math.pow(2, -10 * x) * math.sin((0.5 * x - 0.075) * (2 * math.pi) / 0.3) + 1
ElasticOutQuarter = lambda x: math.pow(2, -10 * x) * math.sin((0.025 * x - 0.075) * (2 * math.pi) / 0.3) + 1
ElasticInOut = lambda x: ToInOut(ElasticIn, x)
```

```
numToEasing = {
    0: Linear,
    1: QuadIn, 2: QuadOut,
    3: QuadIn, 4: QuadOut, 5: QuadInOut,
    6: CubicIn, 7: CubicOut, 8: CubicInOut,
    9: QuartIn,  10: QuartOut, 11: QuartInOut,
    12: QuintIn, 13: QuintOut, 14: QuintInOut,
    15: SineIn, 16: SineOut, 17: SineInOut,
    18: ExpoIn, 19: ExpoOut, 20: ExpoInOut,
    21: CircIn, 22: CircOut, 23: CircInOut,
    24: ElasticIn, 25: ElasticOut, 26: ElasticOutHalf, 27: ElasticOutQuarter, 28: ElasticInOut,
    29: BackIn, 30: BackOut, 31: BackInOut,
    32: BounceIn, 33: BounceOut, 34: BounceInOut
}
```

B.1.2 Object-Oriented Files

B.1.2.1 small length of code

Source: `https://github.com/storborg/packagetrack/blob/e1e5417565b8e2a919936713e1939db5aa895e56/packagetrack/usps.py`

```
class USPSInterface(object):
    def identify(self, tracking_number):
        return (tracking_number.startswith('91') or
                tracking_number.startswith('94'))

    def url(self, tracking_number):
        return ('http://trkcnfrm1.smi.usps.com/PTSInternetWeb/'
                'InterLabelInquiry.do?origTrackNum=%s' % tracking_number)

    def validate(self, tracking_number):
        tracking_num = tracking_number[:-1].replace(' ', '')
        odd_total = 0
        even_total = 0
        for ii, digit in enumerate(tracking_num):
            if ii % 2:
                odd_total += int(digit)
            else:
                even_total += int(digit)
        total = odd_total + even_total * 3
        check = ((total - (total % 10) + 10) - total) % 10
        return (check == int(tracking_number[-1:]))
```

B.1.2.2 medium length of code

Source: `https://github.com/angr/angr-management/blob/b53e85f6f4bcc84c408e56f5cc41efcbfd2496c7/angrmanagement/utils/block_objects.py`

```
class FunctionHeader:
    __slots__ = ('name', 'prototype', 'args', )

    def __init__(self, name, prototype=None, args=None):
        self.name = name
        self.prototype = prototype
        self.args = args


class Variables:
    __slots__ = ['variables']

    def __init__(self, variables):
        self.variables = variables


class PhiVariable(Variables):
    __slots__ = ['variable']
    def __init__(self, variable, variables):
        super().__init__(variables)
        self.variable = variable


class Label:
    __slots__ = ['addr', 'text']

    def __init__(self, addr, text):
        self.addr = addr
        self.text = text
```

B.1.2.3 large length of code

Source: `https://github.com/geemaple/leetcode/blob/68bc5032e1ee52c22ef2f2e608053484c487af54/leetcode/131.palindrome-partitioning.py`

```
class Solution(object):
    def __init__(self):
        self.is_palindrome = None

    def pre_build(self, s):
        size = len(s)
        self.is_palindrome = [[False for _ in range(size)] for _ in range(↩
    size)]

        for t in range(size):
            i = j = t
            while i >= 0 and j < size and s[i] == s[j]:
                self.is_palindrome[i][j] = True
                i -= 1
                j += 1

            i = t
            j = t + 1
```

```
                while i >= 0 and j < size and s[i] == s[j]:
                    self.is_palindrome[i][j] = True
                    i -= 1
                    j += 1

    def partition(self, s):
        self.pre_build(s)
        results = []
        ans = []
        self.helper(s, 0, ans, results)
        return results

    def helper(self, s, start, ans, results):
        if start == len(s):
            results.append(list(ans))

        for i in range(start, len(s)):
            if not self.is_palindrome[start][i]:
                continue

            ans.append(s[start: i + 1])
            self.helper(s, i + 1, ans, results)
            ans.pop()
```

B.1.3 Procedural Files

B.1.3.1 small length of code

Source: `https://github.com/blackbuntu/blackbuntu/blob/23e7ca77dc659c16345af55b7d75c6f430221f98/opt/blackbuntu/vulnerability-analysis/inguma/lib/libasciienc.py`

```
def encrypt(string):
    a = string
    new_string = ''
    for x in a:
        new_string = new_string+str(ord(x))+' '
    return new_string


def unencrypt(string):
    a = string
    new_string = ''
    b = a.split()
    for x in b:
        new_string = new_string+chr(int(x))
    return new_string
```

B.1.3.2 medium length of code

Source: `https://github.com/fanping9540/zefeng/blob/77481b3106830bd1560ce7e7de096324f0a617e7/exercises/change/example.py`

```
def find_fewest_coins(coins, target):
    if target < 0:
        raise ValueError("cannot find negative change values")
    min_coins_required = [1e9] * (target + 1)
    last_coin = [0]*(target + 1)
    min_coins_required[0] = 0
    last_coin[0] = -1
    for change in range(1, target + 1):
        final_result = min_coins_required[change]
        for coin in coins:
            if coin <= change:
                result = min_coins_required[change - coin] + 1
                if result < final_result:
                    final_result = result
                    last_coin[change] = change - coin
        min_coins_required[change] = final_result
    if min_coins_required[target] == 1e9:
        raise ValueError("no combination can add up to target")
    else:
        last_coin_value = target
        array = []
        while(last_coin[last_coin_value] != -1):
            array.append(last_coin_value-last_coin[last_coin_value])
            last_coin_value = last_coin[last_coin_value]
        return array
```

B.1.3.3 large length of code

Source: `https://github.com/IBM-Security/ibmsecurity/blob/22606e5d51bc45808a431f33ec4d2878d7506cb4/ibmsecurity/isam/aac/api_protection/grants_user.py`

```
logger = logging.getLogger(__name__)


def get(isamAppliance, userid, check_mode=False, force=False):
    return isamAppliance.invoke_get("Get grants by userid",
                                    "/iam/access/v8/grants/userIds/{0}".↩
    format(userid))


def get_recent(isamAppliance, userid, timestamp, token_type='refresh_token', ↩
    check_mode=False, force=False):
    ret_obj = get(isamAppliance=isamAppliance, userid=userid)

    recent_tokens = []
    other_tokens = []
    for attrbs in ret_obj['data']:
        for tok in attrbs['tokens']:
            if tok['dateCreated'] > timestamp and (tok['subType'] == ↩
    token_type or token_type is None):
                recent_tokens.append(tok)
            else:
                other_tokens.append(tok)
    new_ret_obj = isamAppliance.create_return_object()
    new_ret_obj['data']['recent'] = recent_tokens
    new_ret_obj['data']['other'] = other_tokens

    return new_ret_obj


def delete(isamAppliance, userid, check_mode=False, force=False):
    if force is True or _check(isamAppliance, userid) is True:
        if check_mode is True:
            return isamAppliance.create_return_object(changed=True)
        else:
            return isamAppliance.invoke_delete("Delete grants by userid",
                                               "/iam/access/v8/grants/userIds↩
/{0}".format(userid))

    return isamAppliance.create_return_object()


def _check(isamAppliance, userid):
    try:
        ret_obj = get(isamAppliance, userid)
        if len(ret_obj['data']) > 0:
            return True
    except:
        pass

    return False
```

B.1.4 Mixed Files

B.1.4.1 small length of code

Source: `https://github.com/pipermerriam/eth-testrpc/blob/328e1ba3dfce6273527773505701522febef79a0/tests/endpoints/`

test_personal_sendTransaction.py

```python
def test_personal_sendTransaction(accounts, rpc_client, password_account, account_password):
    initial_balance = rpc_client('eth_getBalance', [accounts[1]])

    with pytest.raises(AssertionError):
        rpc_client('personal_signAndSendTransaction', [{
            'from': password_account,
            'to': accounts[1],
            'value': 1234,
        }, "incorrect-password"])
    assert rpc_client('eth_getBalance', [accounts[1]]) == initial_balance

    rpc_client('personal_signAndSendTransaction', [{
        'from': password_account,
        'to': accounts[1],
        'value': 1234,
    }, account_password])
    after_balance = rpc_client('eth_getBalance', [accounts[1]])

    assert after_balance - initial_balance == 1234
```

B.1.4.2 medium length of code

Source: https://github.com/asottile/babi/blob/a10ae3c3faed4c4726187caeb4e19f425ab23df3/tests/features/open_test.py

```python
def test_open_cancelled(run, tmpdir):
    f = tmpdir.join('f')
    f.write('hello world')

    with run(str(f)) as h, and_exit(h):
        h.await_text('hello world')

        h.press('^P')
        h.await_text('enter filename:')
        h.press('^C')

        h.await_text('cancelled')
        h.await_text('hello world')


def test_open(run, tmpdir):
    f = tmpdir.join('f')
    f.write('hello world')
    g = tmpdir.join('g')
    g.write('goodbye world')

    with run(str(f)) as h:
        h.await_text('hello world')

        h.press('^P')
        h.press_and_enter(str(g))

        h.await_text('[2/2]')
        h.await_text('goodbye world')

        h.press('^X')
        h.await_text('hello world')

        h.press('^X')
        h.await_exit()
```

B.1.4.3 large length of code

Source: `https://github.com/D0WN3D/gobyteman/blob/bb0682669b6b464efd77649c1c4cb1190220b486/lib/pycoin/pycoin/ecdsa/native/bignum.py`

```
def bignum_type_for_library(library):
    ULONG_FACTOR = 1 << (8 * ctypes.sizeof(ctypes.c_ulong))
```

```python
class BignumType(ctypes.Structure):
    _fields_ = [
        ('d', ctypes.POINTER(ctypes.c_ulong)),
        ('top', ctypes.c_int),
        ('dmax', ctypes.c_int),
        ('neg', ctypes.c_int),
        ('flags', ctypes.c_int),
    ]

    def __init__(self, n=0):
        negative = (n < 0)
        if negative:
            n = -n
        the_len = (n.bit_length() + 7)//8
        sign = b'\x80' if negative else b'\0'
        the_bytes = struct.pack(">L", the_len+1) + sign + to_bytes(n, ↩
the_len, "big")
        library.BN_mpi2bn(the_bytes, the_len + 5, self)

    def __del__(self):
        library.BN_clear_free(self)

    def __int__(self):
        "cast to int"
        return self.as_int()

    def to_int(self):
        value, factor = 0, 1
        for w in self.datawords():
            value += w * factor
            factor *= ULONG_FACTOR
        if self.neg:
            value = -value
        return value

    def datawords(self):
        return (self.d[k] for k in range(self.top))

    def __repr__(self):
        return "BignumType(%d)" % self.to_int()

return BignumType
```

B.2 Task 2 Code Listings: Bug Localization

In this section we list all sub-task code for the bug localization Task 2. Note that these codes were hand-created by the authors, and the bugs were artificially introduced. The bug is the same for a given program, regardless of paradigm.

B.2.1 Cube

The bug is where the value is properly multiplied three times (to compute the cube of the value), but it also then multiplies the result by the constant 3 resulting in $3 * x^3$ instead of x^3.

B.2.1.1 Cube: Object-Oriented

```
# Find a logical bug in the code below
# The following code prints an output for a cube of a list of numbers.

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()
```

B.2.1.2 Cube: Procedural

```
# Find a logical bug in the code below
# The following code prints an output for a cube of a list of numbers.

items = [1, 2, 3, 4]

def cube(x):
    return x * x * x * 3

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

B.2.1.3 Cube: Functional

```
# Find a logical bug in the code below
# The following code prints an output for a cube of a list of numbers.

items = [1, 2, 3, 4]

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

B.2.1.4 Cube: Mixed

```
# Find a logical bug in the code below
# The following code prints an output for a cube of a list of numbers.

items = [1, 2, 3, 4]

for i in items:
    print(i * i * i * 3)
```

B.2.2 Factorial

The bug was that the value returned in the base case was 0 when it should be 1.

B.2.2.1 Factorial: Object-Oriented

```python
# Find a logical bug in the code below
# The following code provides a factorial of a number

import sys
n = int(sys.argv[1])

class Number:
    def __init__(self, n):
        self.n = n

    def factorial(self):
        f = 0
        for i in range(1, self.n + 1):
            f = f * i
        return f

factorial = Number(n).factorial()

print(f'The factorial of {n} is {factorial}')
```

B.2.2.2 Factorial: Procedural

```python
# Find a logical bug in the code below
# The following code provides a factorial of a number

import sys
n = int(sys.argv[1])

def factorial(n):
    return 0 if n <= 1 else n * factorial(n - 1)

print(f'The factorial of {n} is {factorial(n)}')
```

B.2.2.3 Factorial: Functional

```python
# Find a logical bug in the code below
# The following code provides a factorial of a number

import sys
n = int(sys.argv[1])

def recursive_lambda(func):
    def ret(*args):
        return func(ret, *args)
    return ret

factorial = recursive_lambda(lambda factorial, n: n * factorial(n - 1) if n >↩
     1 else 0)(n)

print(f'The factorial of {n} is {factorial}')
```

B.2.2.4 Factorial: Mixed

```python
# Find a logical bug in the code below
# The following code provides a factorial of a number

import sys
n = int(sys.argv[1])

factorial = 0
for i in range(1, n + 1):
   factorial = factorial * i

print(f'The factorial of {n} is {factorial}')
```

B.2.3 Largest Number

The bug is that when finding the maximum number, the comparison is greater than when it should be less than.

B.2.3.1 Largest Number: Object-Oriented

```python
# Find a logical bug in the code below
#This code finds the largest number from a list in Python.

heights = [10, 20, 4, 45, 99]

class MyList:
    def __init__(self, lst):
        self.lst = lst

    def max(self):
         maximum = self.lst[0]

         for x in self.lst:
            if maximum > x:
                 maximum = x

         return maximum

max_height = MyList(heights).max()
print('Largest element is:', max_height)
```

B.2.3.2 Largest Number: Procedural

```python
# Find a logical bug in the code below
#This code finds the largest number from a list in Python.

heights = [10, 20, 4, 45, 99]

def myMax(list1):
     maximum = list1[0]

     for x in list1:
        if maximum > x:
             maximum = x

     return maximum
```

```python
max_height = myMax(heights)
print('Largest element is:', max_height)
```

B.2.3.3 Largest Number: Functional

```python
#Find a logical bug in the code below
#This code finds the largest number from a list in Python.

from functools import reduce

heights = [10, 20, 4, 45, 99]
max_height = reduce(lambda x, y: x if x < y else y, heights)
print('Largest element is:', max_height)
```

B.2.3.4 Largest Number: Mixed

```python
# Find a logical bug in the code below
#This code finds the largest number from a list in Python.

import sys

heights = [10, 20, 4, 45, 99]

def bigger(x, y):
    return y > x

max_height = -sys.maxsize - 1

for x in heights:
    if bigger(x, max_height):
        max_height = x

print('Largest element is:', max_height)
```

B.2.4 Palindrome

The bug was the use of an assignment operator (=) when a comparison operator (==) was intended.

B.2.4.1 Palindrome: Object-Oriented

```python
# Find a logical bug in the code below
# The following code takes an input string and checks for Palindrome.

import sys
word = sys.argv[1]

class Palindrome:
   def __init__(self, word):
      self.word = word.lower()

   def check(self):
      word = self.word.replace(' ', '')
      return word = word[::-1]

print(Palindrome(word).check())
```

B.2.4.2 Palindrome: Procedural

```python
# Find a logical bug in the code below
# The following code takes an input string and checks for Palindrome.

import sys
word = sys.argv[1]

def palindrome(word):
    word = word.lower().replace(' ', '')
    return word = word[::-1]

print(palindrome(word))
```

B.2.4.3 Palindrome: Functional

```python
# Find a logical bug in the code below
# The following code takes an input string and checks for Palindrome.

import sys
word = sys.argv[1]

palindrome = lambda some_str: some_str.lower().replace(' ', '') = some_str.↩
    lower().replace(' ', '')[::-1]

print(palindrome(word))
```

B.2.4.4 Palindrome: Mixed

```python
# Find a logical bug in the code below
# The following code takes an input string and checks for Palindrome.

import sys
word = sys.argv[1]

word = word.lower().replace(' ', '')
rev = reversed(word.casefold())

if list(word) = list(rev):
   print('True')
else:
   print('False')
```

Appendix C

Fixation Tables

In this appendix, we list all fixation counts and durations for every line of code in each of the sub-tasks.

Table C.1: Fixation duration and fixation counts for functional small classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Functional (small)		1	39,145	36
	func,proc	3	98,708	108
	func,proc	5	94,424	91
	func	7	79,896	117
	func,proc	9	45,206	64
	func,oo	11	94,281	93
	func,proc	13	78,930	139
	func,proc	15	133,236	116
	func,proc	17	51,331	71
	func,oo	19	79,936	74
	func,proc	21	52,510	51
	func,oo	23	57,326	47

Table C.2: Fixation duration and fixation counts for functional medium classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Functional (medium)	proc	1	49,660	67
	proc	2	68,579	77
	imp	3	79,088	109
	proc	4	567	1
	proc	5	27,047	43
	proc	6	68,744	79
	proc	9	55,356	63
	proc	10	68,636	79
	imp	11	53,761	80
	imp	12	26,608	52
	proc	14	13,504	22
	proc	15	37,449	27
	imp	16	19,409	37
	imp	17	834	1
	imp	18	29,821	29
	proc	21	30,429	56
	proc	22	46,634	62
	proc	23	28,873	32
	proc	24	26,966	46
	proc	26	23,565	32
	proc	29	45,186	38
	proc	30	29,051	43
	proc	31	22,964	36
	proc	32	123,572	25
	proc	34	13,998	31
	proc	35	7,897	17
	proc	38	116,286	52
	proc	39	157,837	84
	proc	40	44,492	6

Table C.3: Fixation duration and fixation counts for functional large classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Functional (large)	func	1	14,650	24
	func, proc	2	38,280	50
	func	3	7,716	14
	func	4	13,629	23
	func, proc	5	7,817	15
	func	6	3,717	6
	func	7	5,552	13
	func, proc	8	3,783	6
	func, proc	9	10,314	19
	func	10	4,415	8
	func, proc	11	5,988	13
	func, proc	12	6,299	13
	func	13	3,018	9
	func, proc	14	55,430	7
	func	15	8,109	17
	func, oo	17	12,866	24
	func, proc	18	10,483	24
	func	19	65,791	13
	func, oo	21	4,444	9
	func, proc	22	6,696	12
	func, proc	23	5,568	12
	func, proc	25	114,000	7
	func	26	9,245	19
	func, proc	27	14,084	18
	func	29	21,266	9
	func, proc	30	8,736	17
	func	31	9,683	10
	func, oo	33	11,994	21
	func	34	31,522	29
	func	35	6,249	13
	func, proc	36	8,815	15
	func, oo	38	13,131	26
	func	39	80,044	32
	func, oo	40	21,982	46
	func	41	66,359	56
	func	42	36,494	51
	func	44	91,116	65
	func	45	69,369	54
	func	46	22,568	44
	func	47	21,292	35
	func	48	52,131	17
	func	49	12,280	25
	func	50	9,247	14
	func	51	5,977	14
	func	52	8,337	22
	func	53	5,550	16
	func	54	8,416	24
	func	55	9,254	27
	func	56	10,048	13

Table C.4: Fixation duration and fixation counts for mixed small classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Mixed small	proc	1	77,738	86
	proc	2	125,824	136
	proc	4	60,969	73
	proc	5	95,568	109
	proc	6	37,070	79
	proc	7	16,666	35
	oo	8	20,012	26
	oo	9	12,972	29
	oo	10	118,823	92
	oo	12	156,203	69
	proc	13	28,198	54
	proc	14	40,505	32
	oo	15	3,650	7
	proc	16	16,298	36
	proc	17	153,006	69
	proc	19	42,060	44

Table C.5: Fixation duration and fixation counts for mixed meduium classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Mixed medium	oo	1	81,728	102
	proc	2	102,710	148
	proc	3	77,297	131
	oo	5	81,240	105
	func	6	96,822	136
	proc	8	69,728	59
	proc	9	56,196	60
	proc	10	37,990	50
	proc	12	45,362	48
	oo	13	30,600	54
	oo	16	154,630	94
	oo	17	91,279	100
	oo	18	57,098	107
	oo	19	36,573	69
	oo	20	44,756	67
	oo	22	120,542	81
	oo	23	51,905	88
	oo	25	35,377	56
	proc	26	33,144	59
	proc	28	20,784	43
	proc	29	18,643	30
	proc	31	9,650	17
	proc	32	16,440	30
	oo	34	13,896	29
	proc	35	27,781	37

Table C.6: Fixation duration and fixation counts for mixed large classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Mixed large	proc	1	65,797	96
	oo	2	75,971	138
	proc	4	49,216	86
	proc	5	65,313	76
	proc, oo	6	46,389	70
	proc, oo	7	26,350	45
	proc, oo	8	13,876	26
	proc, oo	9	10,917	22
	proc, oo	10	14,502	33
	proc, oo	11	466	1
	proc	13	29,416	56
	oo	14	29,912	50
	proc	15	27,066	54
	proc	16	21,986	34
	proc	17	18,520	43
	proc	18	27,347	67
	proc	19	22,318	50
	proc	20	20,531	40
	proc, oo	22	27,644	46
	proc, oo	23	48,701	46
	proc, oo	25	52,434	43
	proc, oo	26	31,916	42
	proc, oo	27	30,217	41
	proc, oo	29	19,779	32
	proc, oo	30	81,008	45
	proc, oo	31	27,016	53
	proc, oo	32	14,563	27
	proc, oo	33	14,501	27
	proc, oo	34	13,599	23
	proc, oo	35	13,432	17
	proc, oo	36	5,813	14
	proc, oo	38	59,460	63
	proc, oo	39	31,620	30
	proc, oo	41	24,382	30
	proc, oo	42	23,709	39
	proc, oo	44	18,636	29

Table C.7: Fixation duration and fixation counts for procedural small classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Procedural small	proc	1	91,376	44
	proc	2	82,691	81
	proc	3	47,093	77
	proc	4	56,281	70
	proc	5	175,313	115
	proc	6	69,151	63
	proc	9	45,238	62
	proc	10	117,756	64
	proc	11	64,204	51
	proc	12	147,441	55
	proc	13	79,372	38
	proc	14	68,057	63
	proc	15	61,275	30

Table C.8: Fixation duration and fixation counts for procedural medium classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Procedural medium	proc	1	74,351	82
	proc	2	48,303	80
	proc	3	47,096	76
	proc	4	41,201	65
	proc	5	45,137	57
	proc	6	27,413	59
	proc	7	16,829	34
	proc	8	35,107	64
	proc	9	53,138	76
	proc	10	25,994	49
	proc	11	32,125	48
	proc	12	49,487	75
	proc	13	16,103	34
	proc	14	83,725	27
	proc	15	34,044	48
	proc	16	32,466	61
	proc	17	49,323	84
	proc	18	62,885	61
	proc	19	18,354	21
	proc	20	38,492	35
	proc	21	58,490	32
	proc	22	30,249	35
	proc	23	19,430	35
	proc	24	40,584	32
	proc	25	8,419	15

Table C.9: Fixation duration and fixation counts for procedural large classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Procedural large	proc	1	30,450	39
	proc	4	104,289	48
	proc	5	61,827	120
	proc	6	56,588	97
	proc	9	31,062	49
	proc	10	54,685	100
	proc	12	11,484	27
	proc	13	11,919	20
	proc	14	32,324	52
	proc	15	108,531	72
	proc	16	50,480	104
	proc	17	25,470	38
	proc	18	4,635	14
	proc	19	10,515	25
	proc	21	22,920	36
	proc	22	41,989	59
	proc	23	17,435	32
	proc	25	16,737	29
	proc	28	33,210	64
	proc	29	63,460	72
	proc	30	29,125	48
	proc	31	29,246	63
	proc	32	12,217	23
	proc	33	21,471	42
	proc	34	10,832	28
	proc	36	13,262	28
	proc	39	80,934	55
	proc	40	20,670	30
	proc	41	19,066	42
	proc	42	115,046	42
	proc	43	20,657	45
	proc	44	82,597	35
	proc	45	63,593	32
	proc	47	20,529	26

Table C.10: Fixation duration and fixation counts for object-oriented small classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Object-oriented small	oo	1	51,938	55
	oo	2	79,537	139
	oo	3	67,790	109
	oo	4	72,314	77
	oo	6	68,267	75
	oo	7	39,087	76
	oo	8	63,460	53
	oo	10	60,810	85
	oo	11	64,467	92
	oo	12	15,512	34
	oo	13	28,810	44
	oo	14	41,925	74
	oo	15	55,976	37
	oo	16	22,284	36
	oo	17	13,814	18
	oo	18	30,098	44
	oo	19	24,977	46
	oo	20	55,952	49
	oo	21	48,941	38

Table C.11: Fixation duration and fixation counts for object-oriented medium classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Object-oriented medium	oo	1	32,558	22
	oo	2	34,446	48
	oo	4	47,225	48
	oo	5	74,723	47
	oo	6	15,391	31
	oo	7	10,930	19
	oo	10	17,335	26
	oo	11	27,761	47
	oo	13	130,349	46
	oo	14	78,601	45
	oo	17	47,834	52
	oo	18	21,551	27
	oo	20	43,440	47
	oo	21	19,869	40
	oo	22	13,196	12
	oo	25	124,646	41
	oo	26	68,821	37
	oo	28	20,703	26
	oo	29	17,647	14
	oo	30	37,846	10

Table C.12: Fixation duration and fixation counts for object-oriented large classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Object-oriented large	oo	1	8,317	19
	oo	2	40,024	64
	oo	3	65,502	99
	oo	5	34,660	65
	oo	6	50,542	98
	oo	7	87,024	168
	oo	9	53,310	70
	oo	10	33,493	55
	oo	11	60,588	110
	oo	12	57,813	78
	oo	13	37,882	40
	oo	14	38,933	27
	oo	16	2,597	9
	oo	17	14,080	31
	oo	18	66,895	88
	oo	19	33,581	48
	oo	20	24,981	25
	oo	21	10,818	21
	oo	23	81,647	88
	oo	24	36,806	85
	oo	25	23,492	34
	oo	26	34,116	46
	oo	27	50,327	64
	oo	28	21,447	29
	oo	30	54,549	76
	oo	31	77,761	49
	oo	32	30,294	37
	oo	34	97,186	50
	oo	35	17,997	43
	oo	36	6,448	15
	oo	38	72,710	22
	oo	39	17,201	31
	oo	40	14,747	12

Table C.13: Fixation duration and fixation counts for cube functional debugging tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Cube Functional	func	1	6,922	19
	func	2	42,563	45
	func	4	52,643	56
	func	6	455,755	223

Table C.14: Fixation duration and fixation counts for cube mixed debugging tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Cube Mixed		1	10,735	23
		2	26,730	43
	func	4	121,655	109
	proc, oo	6	107,728	76
	proc	7	231,416	110

Table C.15: Fixation duration and fixation counts for cube procedural debugging tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Cube Procedural	proc	1	6,015	10
	proc	2	16,394	29
	proc	4	28,258	30
	proc	6	97,207	48
	proc	7	265,332	140
	proc	9	115,660	64
	proc	10	83,201	101
	proc	11	42,416	29

Table C.16: Fixation duration and fixation counts for cube object-oriented debugging tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Cube Object-oriented	oo	1	3,918	9
	oo	2	17,636	28
	oo	4	19,097	24
	oo	6	8,696	17
	oo	7	55,848	51
	oo	8	38,002	43
	oo	10	15,347	24
	oo	11	167,782	118
	oo	13	20,365	38
	oo	14	42,176	48
	oo	15	53,228	19

Table C.17: Fixation duration and fixation counts for factorial functional classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Factorial Functional	func	1	6,301	15
	func	2	20,029	44
	func	4	12,184	13
	func	5	77,856	88
	func	7	135,303	191
	func	8	193,672	259
	func	9	215,683	271
	func	10	91,626	77
	func	12	771,319	1041
	func	14	107,341	157

Table C.18: Fixation duration and fixation counts for factorial mixed classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Factorial Mixed		1	4,549	11
		2	13,451	24
		4	15,449	11
		5	37,713	38
	proc	7	63,242	58
	func, proc	8	156,134	132
	func	9	198,696	209
	proc	11	114,225	95

Table C.19: Fixation duration and fixation counts for factorial procedural classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Factorial Procedural	proc	1	3,149	4
	proc	2	11,369	11
	proc	4	9,082	19
	proc	5	78,358	83
	proc	7	102,462	60
	proc	8	379,710	287
	proc	10	223,018	135

Table C.20: Fixation duration and fixation counts for factorial object-oriented classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Factorial Object-oriented	oo	1	8,095	16
	oo	2	16,164	28
	oo	4	12,790	17
	oo	5	40,831	45
	oo	7	24,076	41
	oo	8	72,786	76
	oo	9	55,666	79
	oo	11	61,556	94
	oo	12	69,015	59
	oo	13	233,094	279
	oo	14	185,162	121
	oo	15	21,508	35
	oo	17	55,245	81
	oo	19	51,433	53

Table C.21: Fixation duration and fixation counts for largest functional classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Largest Functional	func	1	7,383	14
	func	2	9,901	22
	func	4	83,775	47
	func	6	51,296	60
	func	7	225,864	296
	func	8	402,576	343

Table C.22: Fixation duration and fixation counts for largest mixed classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Largest Mixed		1	10,644	19
		2	18,521	36
		4	601	2
		6	48,770	63
		8	80,286	76
	proc	9	126,921	117
	proc, oo	11	214,230	239
	proc,func	13	95,887	97
	proc	14	134,878	184
		15	43,512	63
	proc	17	33,601	28

Table C.23: Fixation duration and fixation counts for largest procedural classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Largest Procedural	proc	1	3,701	10
	proc	2	5,952	12
	proc	4	31,031	55
	proc	6	34,098	58
	proc	7	98,721	138
	proc	9	98,312	109
	proc	10	180,853	130
	proc	11	100,413	49
	proc	13	21,043	23
	proc	15	17,478	27
	proc	16	19,115	6

Table C.24: Fixation duration and fixation counts for largest object-oriented classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Largest Object-oriented	oo	1	6,249	12
	oo	2	13,648	26
	oo	4	9,916	19
	oo	6	15,389	19
	oo	7	38,761	57
	oo	8	43,260	65
	oo	10	28,578	40
	oo	11	61,069	88
	oo	13	182,037	226
	oo	14	206,295	191
	oo	15	102,959	102
	oo	17	64,212	37
	oo	19	38,761	66
	oo	20	35,482	43

Table C.25: Fixation duration and fixation counts for palindrome functional classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Palindrome Functional	func	1	5,501	16
	func	2	39,729	65
	func	4	31,442	27
	func	5	71,708	68
	func	7	653,840	726
	func	9	53,843	72

Table C.26: Fixation duration and fixation counts for classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Palindrome Mixed		1	1,267	5
		2	24,503	26
		4	8,547	15
		5	47,339	54
	func	7	109,676	118
	func	8	279,678	330
	proc	10	233,610	177
	proc	11	109,420	118
	proc	12	27,507	9
		13	20,898	11

Table C.27: Fixation duration and fixation counts for palindrome procedural classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Palindrome Procedural	proc	1	1,699	4
	proc	2	7,565	17
	proc	4	6,013	7
	proc	5	28,929	28
	proc	7	81,360	71
	proc	8	128,966	156
	proc	9	312,792	274
	proc	11	192,826	63

Table C.28: Fixation duration and fixation counts for palindrome object-oriented classification tasks

Task	classification	Line Number	Fixation Duration (ms)	Fixation Counts
Palindrome Object-oriented	oo	1	2,231	5
	oo	2	19,083	39
	oo	4	4,249	9
	oo	5	36,988	50
	oo	7	21,614	36
	oo	8	60,544	89
	oo	9	180,410	183
	oo	11	82,309	64
	oo	12	327,521	316
	oo	13	303,864	263
	oo	15	129,416	108

Bibliography

[1] TIOBE Software BV, *TIOBE Index for August 2021*, `https://tiobe.com/tiobe-index/`, 2021.

[2] P. Carbonnelle, *PYPL PopularitY of Programming Language*, `https://pypl.github.io/`, 2021.

[3] Github, *The top programming languages*, `https://octoverse.github.com/2022/top-programming-languages/`, 2022.

[4] R. Dyer and J. Chauhan, "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*, ACM, Nov. 2022. DOI: `10.1145/3540250.3549158`.

[5] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 422–431, ISBN: 9781467330763.

[6] H. Rajan, T. N. Nguyen, R. Dyer, and H. A. Nguyen, *Boa website*, `http://boa.cs.iastate.edu/boa/`, 2021.

[7] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in Python programs," in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, 2016, pp. 18–23. DOI: `10.1109/SATE.2016.10`.

[8] M. R. Rahman, A. Rahman, and L. Williams, “Share, but be aware: Security smells in Python gists,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 536–540. DOI: 10.1109/ICSME.2019.00087.

[9] R. M. Siegfried, D. Liporace, and K. G. Herbert-Berger, “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*, ser. SIGCSE ’19, Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 1256–1257, ISBN: 9781450358903. DOI: 10.1145/3287324.3293830.

[10] A. Eghbali and M. Pradel, “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*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 760–771, ISBN: 9781450394130. DOI: 10.1145/3540250.3549126.

[11] T. Kohn, G. van Rossum, G. B. Bucher II, Talin, and I. Levkivskyi, “Dynamic pattern matching with Python,” in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 85–98, ISBN: 9781450381758. DOI: 10.1145/3426422.3426983.

[12] V. Romanov, “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*, ser. SPLASH

Companion 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 25–27, ISBN: 9781450381796. DOI: 10.1145/3426430.3428135.

[13] C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall, and G. Robles, "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*, ser. Onward! 2018, Boston, MA, USA: Association for Computing Machinery, 2018, pp. 1–11, ISBN: 9781450360319. DOI: 10.1145/3276954.3276960.

[14] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "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*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 691–701, ISBN: 9781450371216. DOI: 10.1145/3377811.3380352.

[15] R. W. Floyd, "The paradigms of programming," *Commun. ACM*, vol. 22, no. 8, pp. 455–460, Aug. 1979, ISSN: 0001-0782. DOI: 10.1145/359138.359140.

[16] A. T. Duchowski and A. T. Duchowski, *Eye tracking methodology: Theory and practice*. Springer, 2017.

[17] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2015, pp. 96–103. DOI: 10.1109/APSEC.2015.53.

[18] M. A. Just and P. A. Carpenter, "A theory of reading: From eye fixations to comprehension.," *Psychological review*, vol. 87, no. 4, p. 329, 1980.

[19] C. S. Peterson, J. A. Saddler, N. M. Halavick, and B. Sharif, "A gaze-based exploratory study on the information seeking behavior of developers on stack overflow," in *Extended Abstracts of the 2019 CHI Conference on Human Factors*

in Computing Systems, ser. CHI EA '19, Glasgow, Scotland Uk: Association for Computing Machinery, 2019, pp. 1–6, ISBN: 9781450359719. DOI: 10.1145/3290607.3312801.

[20] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, IEEE, 2016, pp. 563–573. DOI: 10.1109/SANER.2016.112.

[21] S. S. Brilliant and T. R. Wiseman, "The first programming paradigm and language dilemma," in *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 338–342, ISBN: 089791757X. DOI: 10.1145/236452.236572.

[22] P. M. Uesbeck, C. S. Peterson, B. Sharif, and A. Stefik, "A randomized controlled trial on the effects of embedded computer language switching," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 410–420, ISBN: 9781450370431. DOI: 10.1145/3368089.3409701.

[23] P. Mayer and A. Bauer, "An empirical analysis of the utilization of multiple programming languages in open source projects," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '15, Nanjing, China: Association for Computing Machinery, 2015, ISBN: 9781450333504. DOI: 10.1145/2745802.2745805.

[24] P. Chakraborty, R. Shahriyar, A. Iqbal, and G. Uddin, "How do developers discuss and support new programming languages in technical q&a site? an empirical study of Go, Swift, and Rust in Stack Overflow," *Information and Software Technology*, vol. 137, p. 106 603, 2021, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106603.

[25] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic, "Developer reading behavior while summarizing java methods: Size and context matters," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 384–395, 2019.

[26] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. K. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[27] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing software developers' eyes and interactions for change tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 202–213, ISBN: 9781450336758. DOI: 10.1145/2786805.2786864.

[28] T. Busjahn, R. Bednarik, A. Begel, *et al.*, "Eye movements in code reading: Relaxing the linear order," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC '15, Florence, Italy: IEEE Press, 2015, pp. 255–265.

[29] M. B. Wells and B. L. Kurtz, "Teaching multiple programming paradigms: A proposal for a paradigm general pseudocode," in *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE

'89, Louisville, Kentucky, USA: Association for Computing Machinery, 1989, pp. 246–251, ISBN: 0897912985. DOI: 10.1145/65293.71222.

[30] T. Bunkerd, D. Wang, R. G. Kula, *et al.*, "How do contributors impact code naturalness? an exploratory study of 50 Python projects," in *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2019, pp. 7–75. DOI: 10.1109/IWESEP49350.2019.00010.

[31] Y. Peng, Y. Zhang, and M. Hu, "An empirical study for common language features used in Python projects," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 24–35. DOI: 10.1109/SANER50967.2021.00012.

[32] N. Bafatakis, N. Boecker, W. Boon, *et al.*, "Python coding style compliance on Stack Overflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 210–214. DOI: 10.1109/MSR.2019.00042.

[33] D. S. Foundation, *Django: The web framework for perfectionists with deadlines.* https://www.djangoproject.com/, 2021.

[34] V. Freitas, *Class-based views vs. function-based views*, https://simpleisbetterthancomplex.com/article/2017/03/21/class-based-views-vs-function-based-views.html, 2021.

[35] A. M. Kuchling, *Functional Programming HOWTO*, https://docs.python.org/3/howto/functional.html, 2021.

[36] V. Zyrianov, D. T. Guarnera, C. S. Peterson, B. Sharif, and J. I. Maletic, "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)*, 2020, pp. 464–475. DOI: 10.1109/ICSME46990.2020.00051.

[37] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi, “Capturing software traceability links from developers’ eye gazes,” in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 201–204, ISBN: 9781450328791. DOI: 10.1145/2597008.2597795.

[38] R. Andersson, L. Larsson, K. Holmqvist, M. Stridh, and M. Nyström, “One algorithm to rule them all? an evaluation and discussion of ten eye movement event-detection algorithms,” *Behavior research methods*, vol. 49, no. 2, pp. 616–637, 2017. DOI: 10.3758/s13428-016-0738-9.

[39] A. Begel and T. Zimmermann, “Analyze this! 145 questions for data scientists in software engineering,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 12–23, ISBN: 9781450327565. DOI: 10.1145/2568225.2568233.