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

Committee:
Dr. Robert Dyer
Dr. Bonita Sharif
Dr. Rahul Purandare

Jigyasa Chauhan
Thesis Defense

IN OUR GRIT, OUR GLORY™

Python is a multi-paradigm programming language and currently one of the **most popular** languages.

Imperative	assignment, logical operations, …
Procedural	def, return, calls, …
Object-Oriented	class, inheritance, with, …
Functional	lambda, for loops, iterators, …

Related Work

- Peng et al. found that developers **choose** certain features more than others to perform a task with Python project (2021)[3]
- Floyd et al. found that if a developer learns imperative style of coding, then it is **harder to switch** to OO paradigm (1979)[5]
- Shrestha et al. found that **prior learning** of a previous language can hinder the grasping of a new language (2020)[4]
- Alexandru et al. studied how developers **lack an understanding** of Python definitions and use them over GitHub and StackOverflow (2018)[2]

Previous Work

- Dyer and Chauhan 2022, explored over 100K+ Python projects from GitHub and classified Python paradigms using a query. They found that functional paradigm was used significantly less than procedural and object-oriented paradigms[1]

- Therefore, we were interested to investigate how Python developers classify Python paradigms.

Motivation

Help the people in education sector by **teaching and training developers** with Python paradigms and features.

Training Python developers with new **language paradigms and features**, required to perform certain tasks in the industry.

Researchers trying to understand **Program Comprehension**. Also, the **Python community** that has been maintaining Python, can understand how **developers use** Python features.

RESEARCH QUESTIONS

Research Questions

CLASSIFICATION

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

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

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

BUG LOCALIZATION

4. Does the predominant paradigm affect how long developer’s take to debug logical errors?

5. Does the predominant paradigm affect a developer’s ability to debug logical errors?

Solution

We needed to investigate developer's behavior with collecting ***surveys*** *and* ***analyzing time data*** *using Python libraries.*

We needed ***eye-tracking*** *to understand developer's behavior with respect to classification and bug localization Python paradigms.*

We needed to ***interview*** *Python developers to understand their approach and methodology towards Python paradigms.*

STUDY DESIGN AND APPROACH

- Boa to search for tasks
- Eclipse IDE for viewing Python code
- Tobii TX300 eye tracker (60 Hz)
- iTrace plugin and toolkit

Participants

- **29+2 participants**
  (removed 1 due to no Python experience and
  1 due to poor eye tracking calibration)

- More than 85% were CS majors
- All participants had at least 1 year of experience in Python

Task Category 1 – Classification of paradigms

- Small code (1-15 statements)
- Medium code (16-30 statements)
- Large code (31-45 statements)

Task Category 2 – Bug localization in different paradigms

- Cube of a number
- Factorial of a number
- Largest number
- Palindrome number

Data Collection Methods

Survey + Task questionnaires (Google forms)

Eye-tracking data (XML files and database by iTrace toolkit)

Audio only interview (audio files → transcribed text on index cards)

Study Flow

1. Pre-questionnaire
2. Training
3. Task Category 1 (Classification)
4. Interview
5. Task Category 2 (Bug Localization)
6. Interview
7. Post-questionnaire

Training Example for Classification (Task 1)

```
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
```

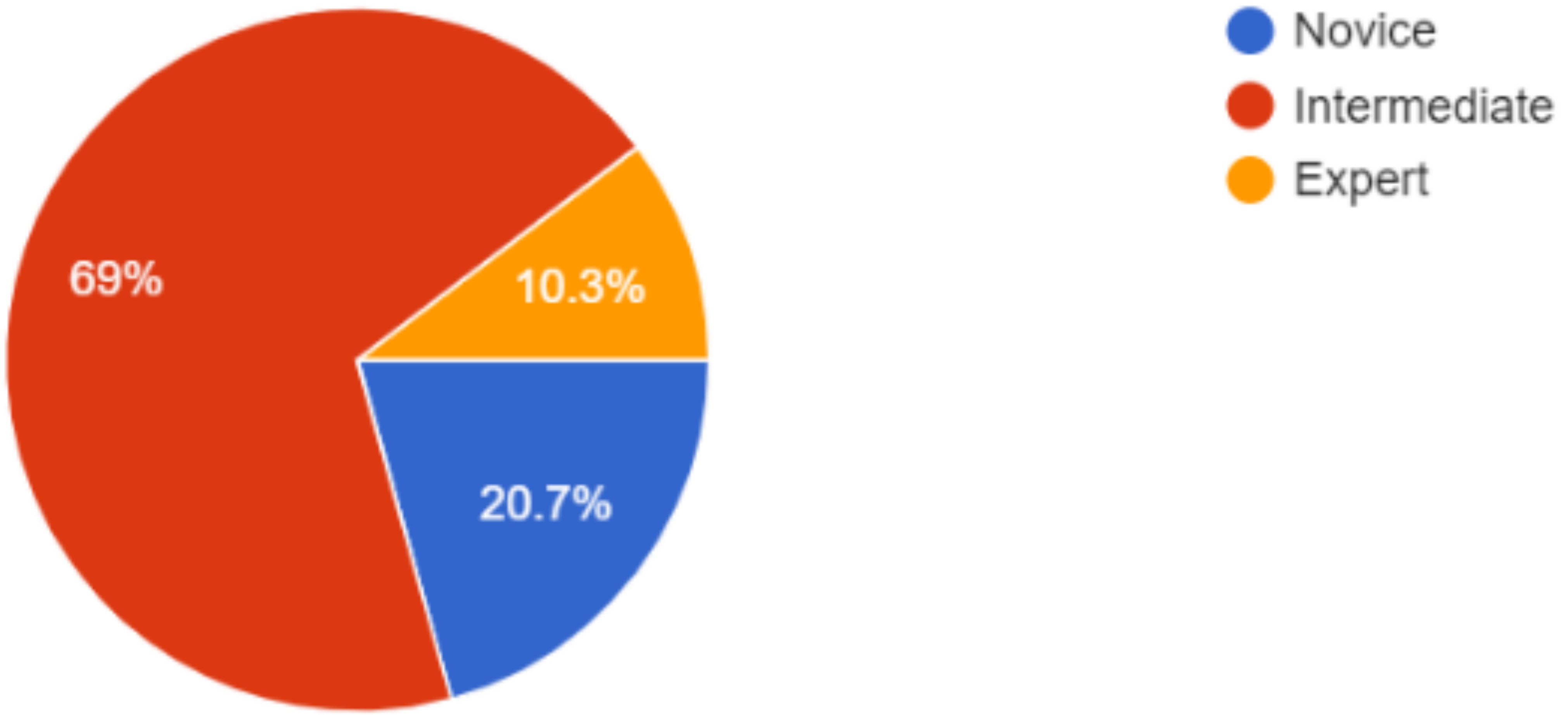
(code listing taken from [1])

Paradigms

func: functional
oo: object-oriented
imp: imperative
proc: procedural

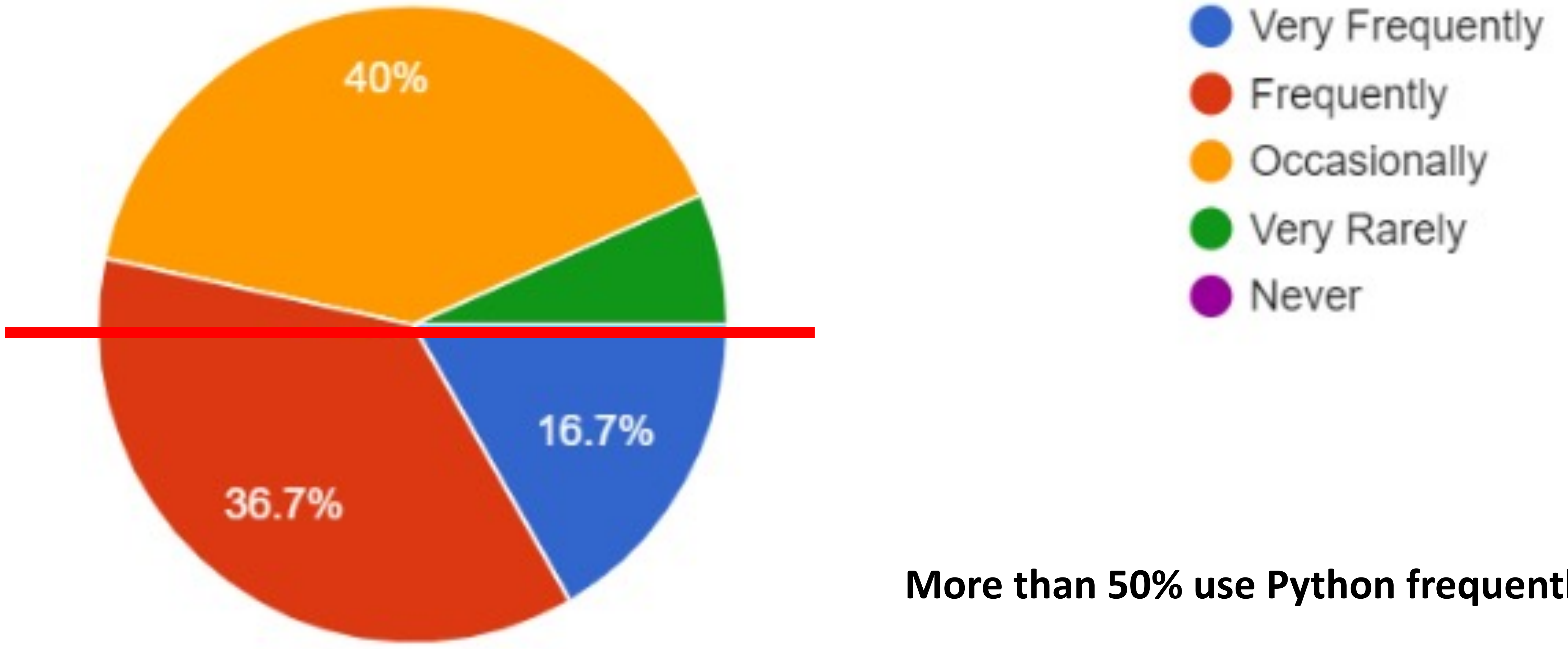
RESULTS: Qualitative

Post-questionnaire Data



How would you rate your programming in Python?

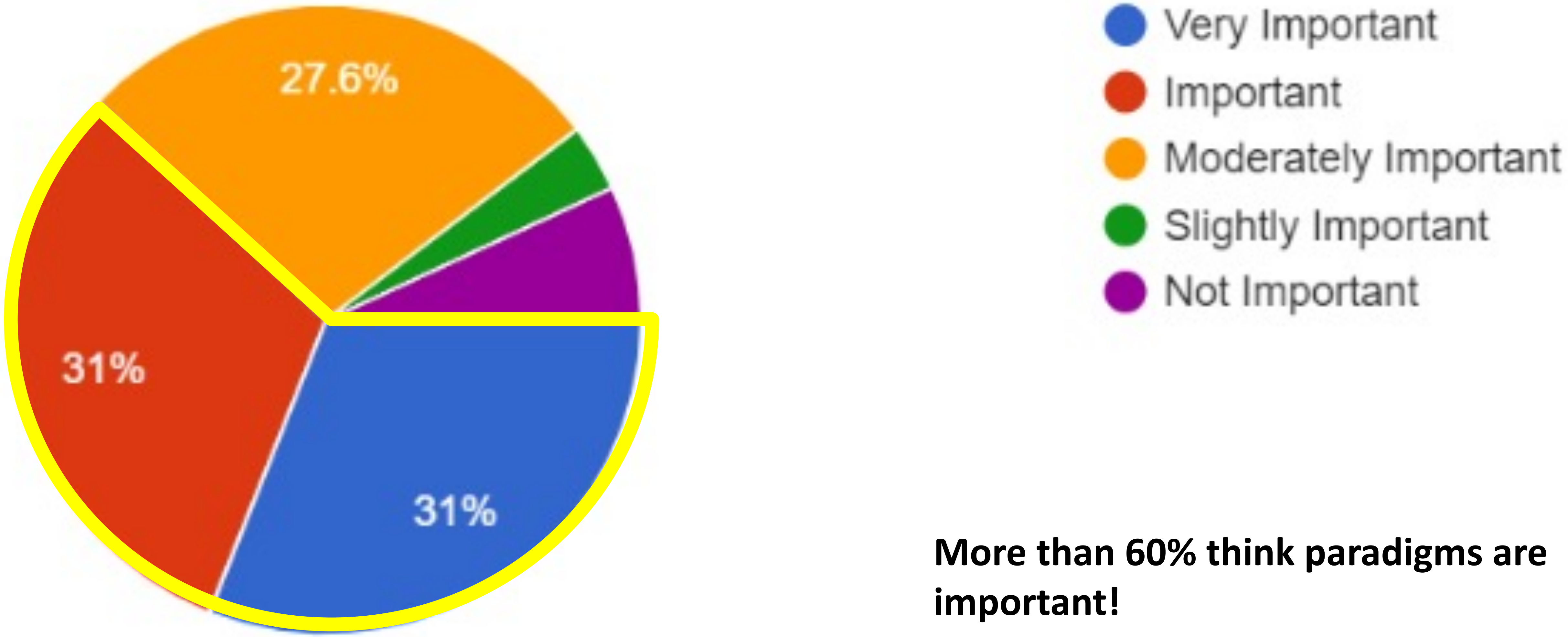
Pre-questionnaire Data



More than 50% use Python frequently!

How often do you program in Python?

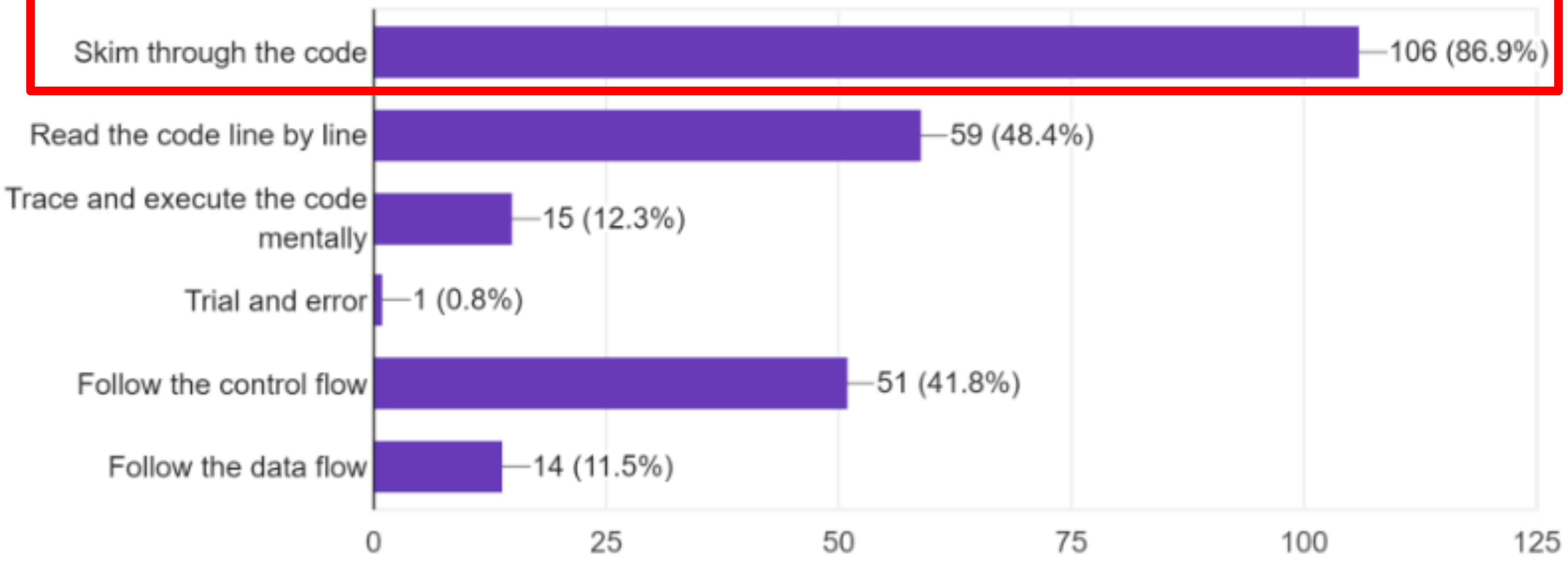
Post-questionnaire Data



More than 60% think paradigms are important!

How important is it for you to code in a specific programming paradigm?
For example: Functional, Object oriented, Procedural

Task 1 Questionnaire



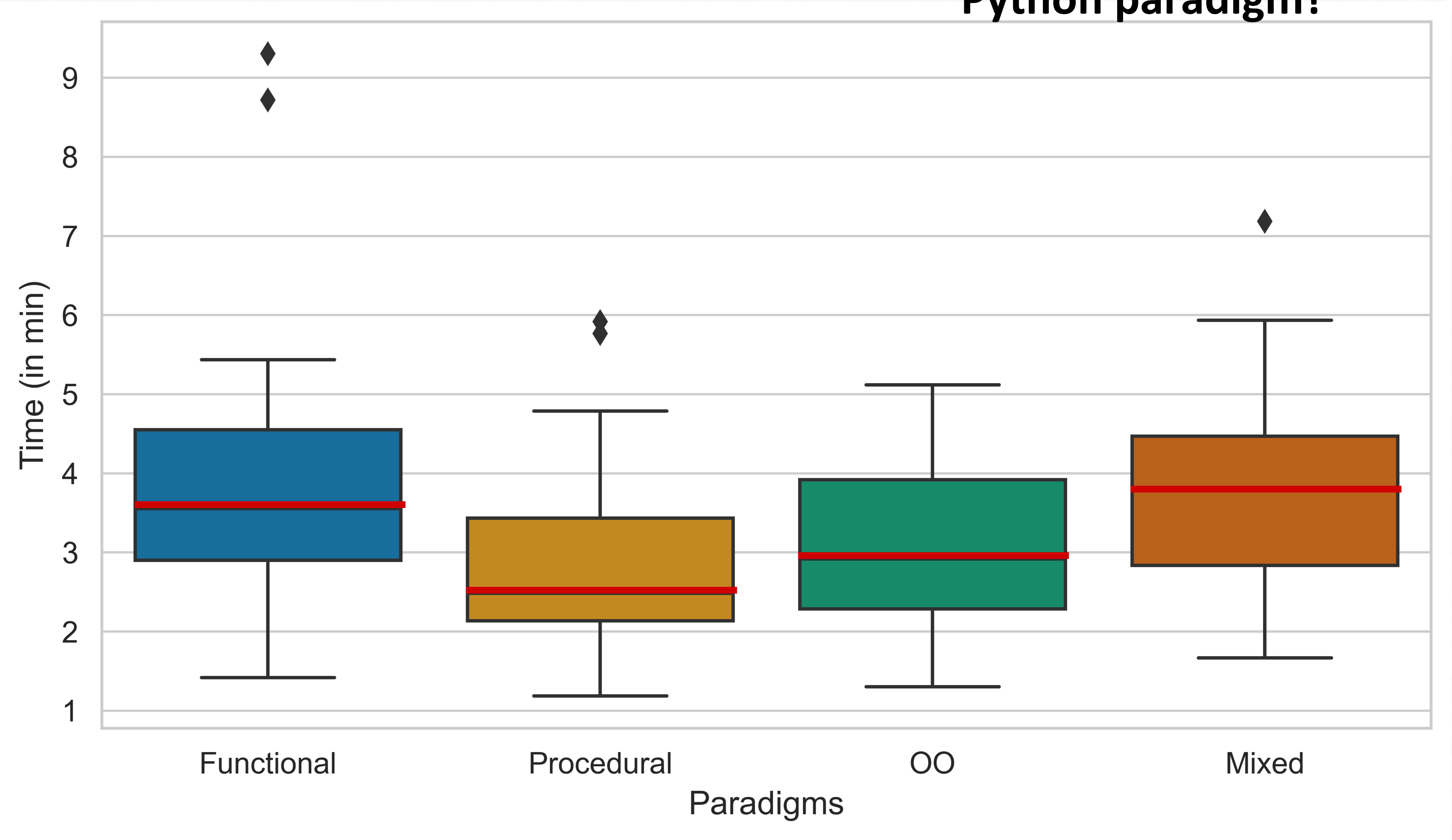
Approach	Functional	Procedural	Object-Oriented	Mixed
Skim	27	25	28	23
Read line by line	10	11	14	18
Control flow	11	14	13	10
Data flow	5	2	3	4
Trace and execute	4	6	2	3
Trial and Error	0	0	0	1

Self-identified approach used to classify the predominant paradigm

RESULTS: Quantitative

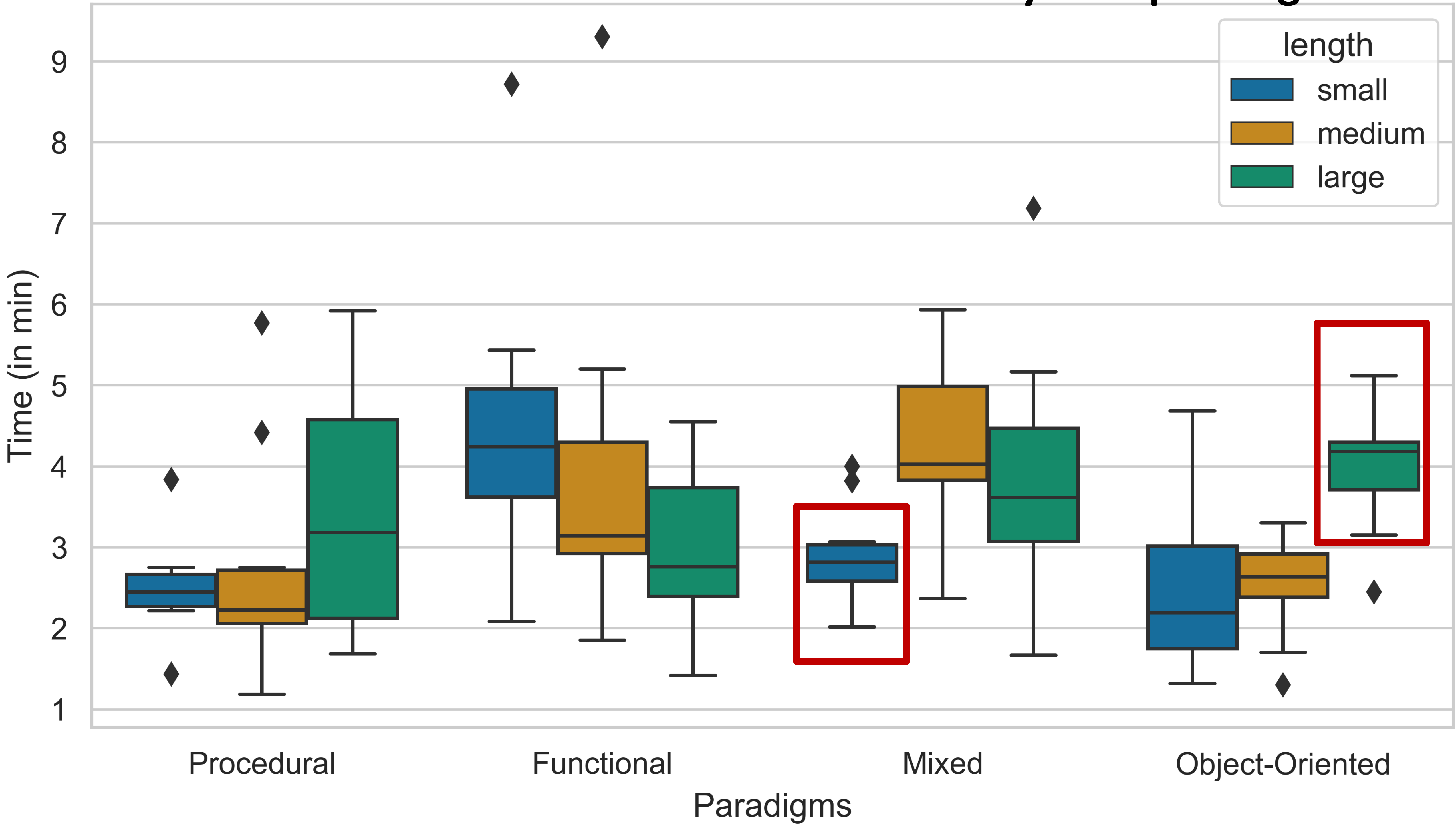
Results: Task Category 1 (Classification)

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



Time taken to classify predominant paradigm for Task 1

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

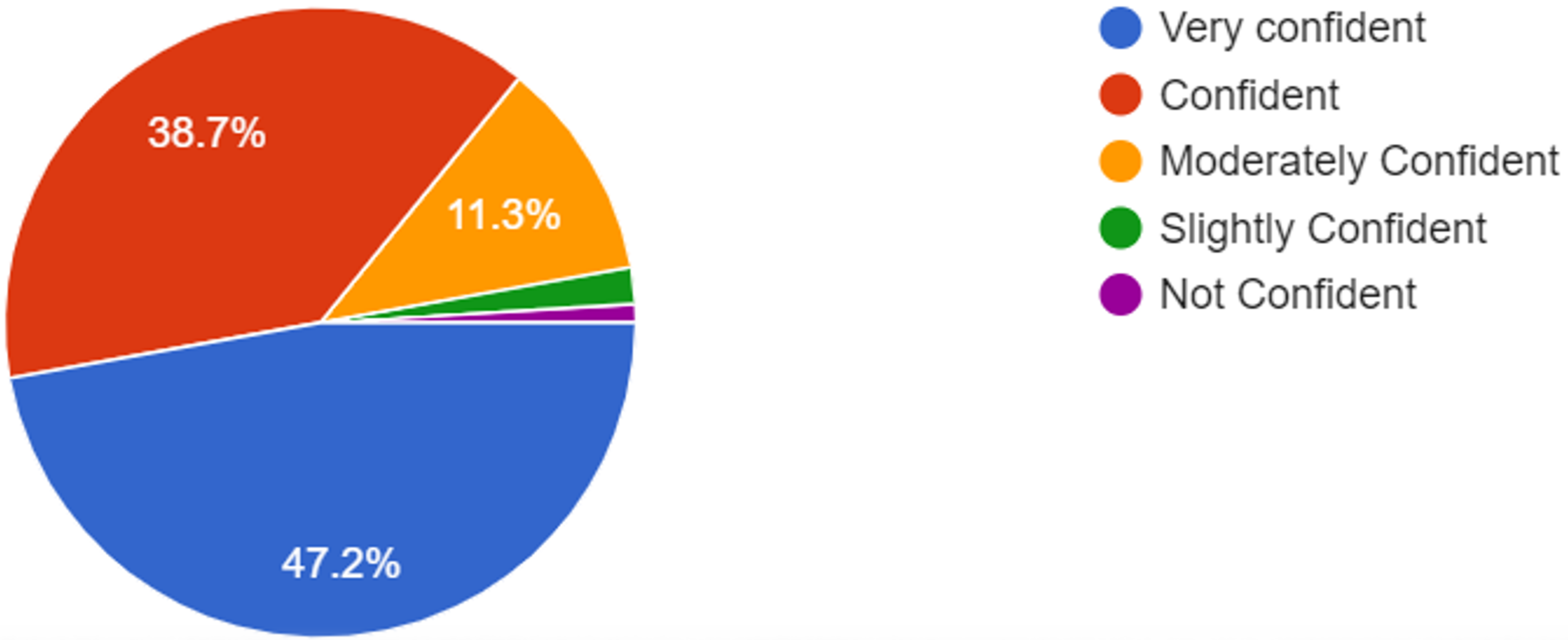


Time taken to classify predominant paradigm for Task 1 by size: small, medium, large

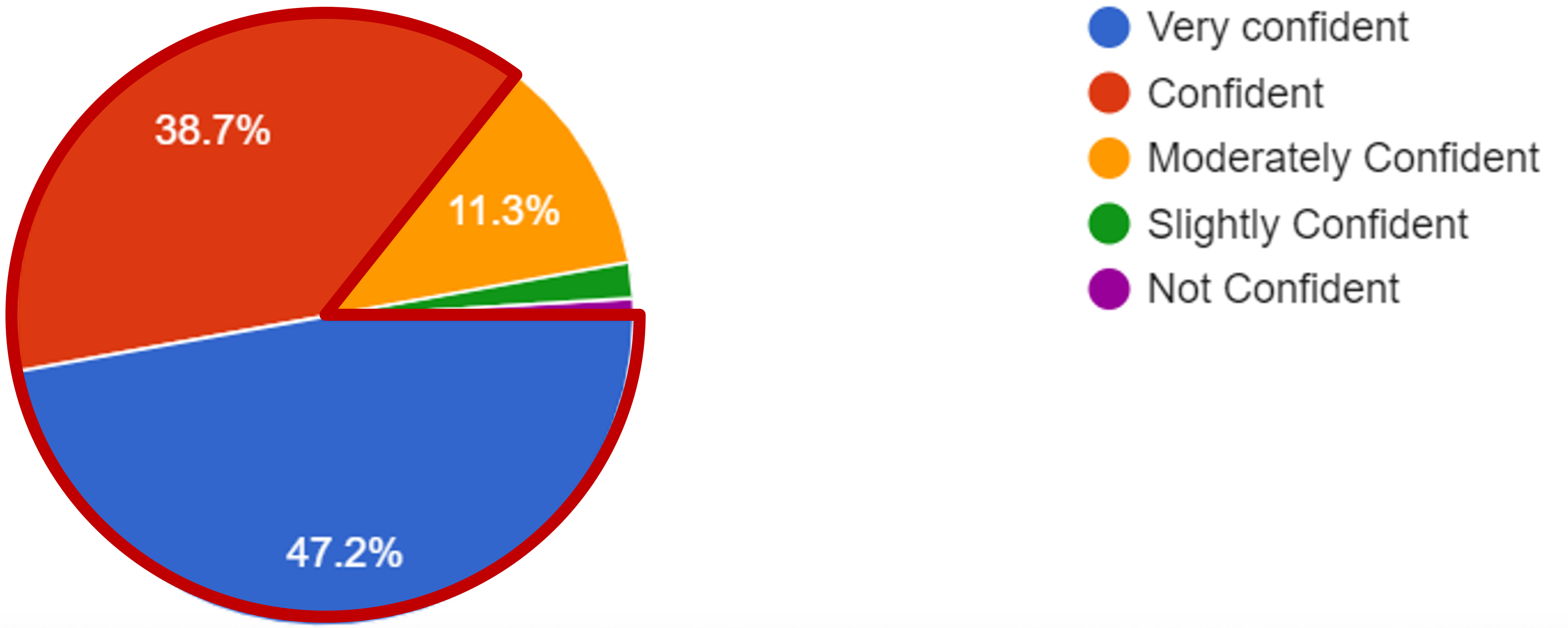
RQ1 Result Summary

- *We found that participants classify all paradigms in a **similar time**.*
- *We see **no correlation** between different length of the code and time taken to classify.*

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



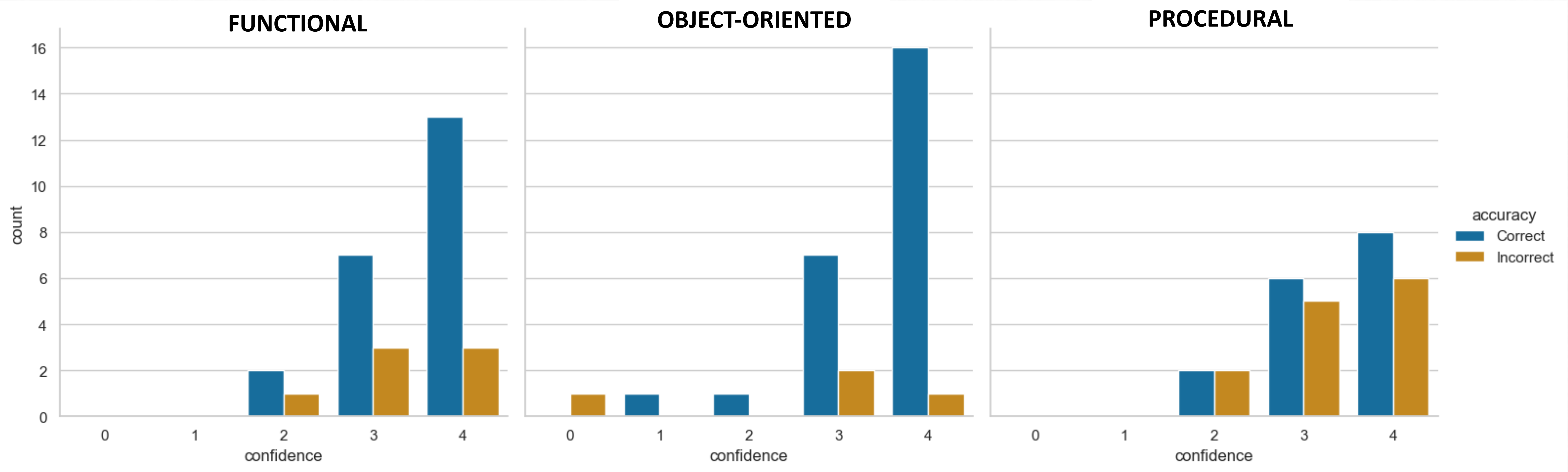
RQ2: How accurately do developers classify the predominant paradigm in Python code?



More than 85% were confident!

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

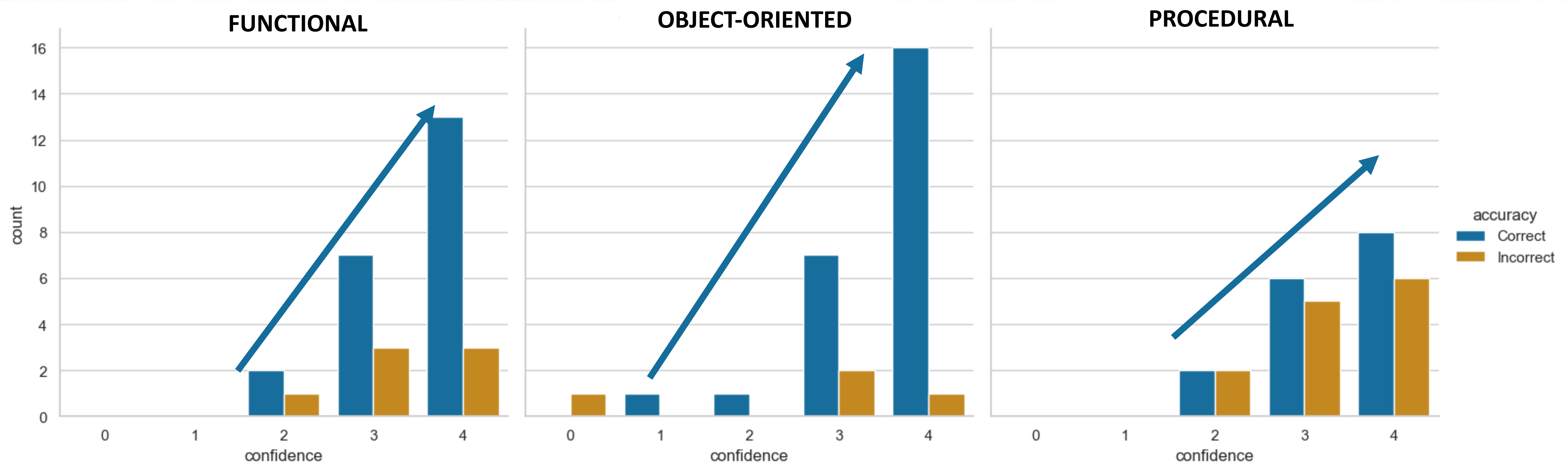
Task 1 – Judgements vs Confidence Levels



Not Confident: 0 Slightly Confident: 1 Moderately Confident: 2 Confident: 3 Very Confident: 4

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

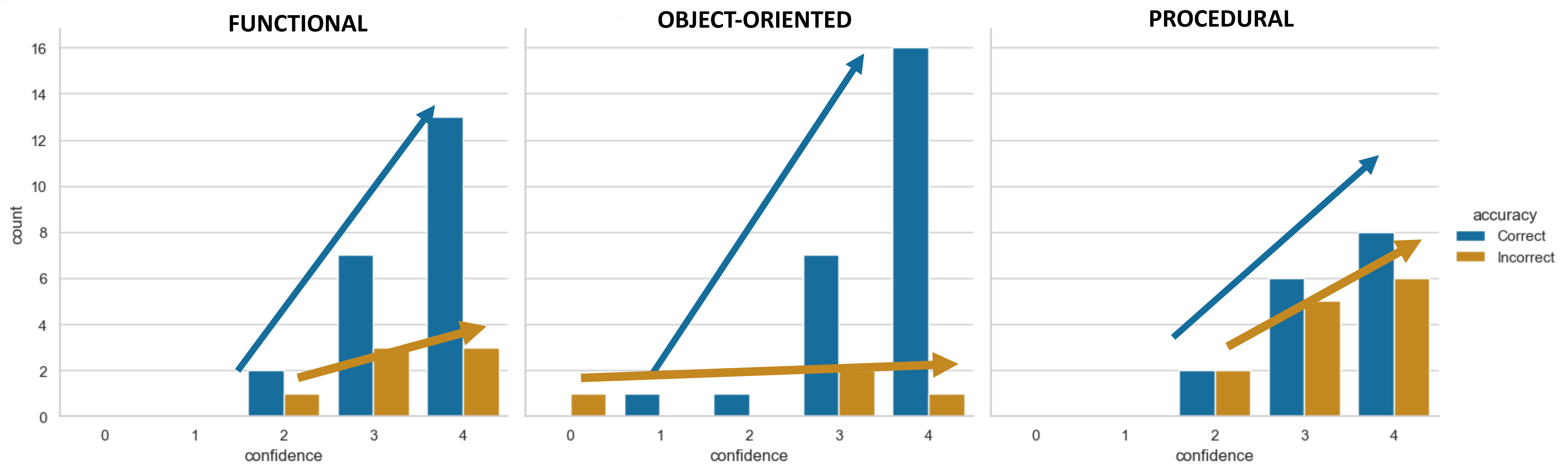
Task 1 – Judgements vs Confidence Levels



Not Confident: 0 Slightly Confident: 1 Moderately Confident: 2 Confident: 3 Very Confident: 4

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

Task 1 – Judgements vs Confidence Levels



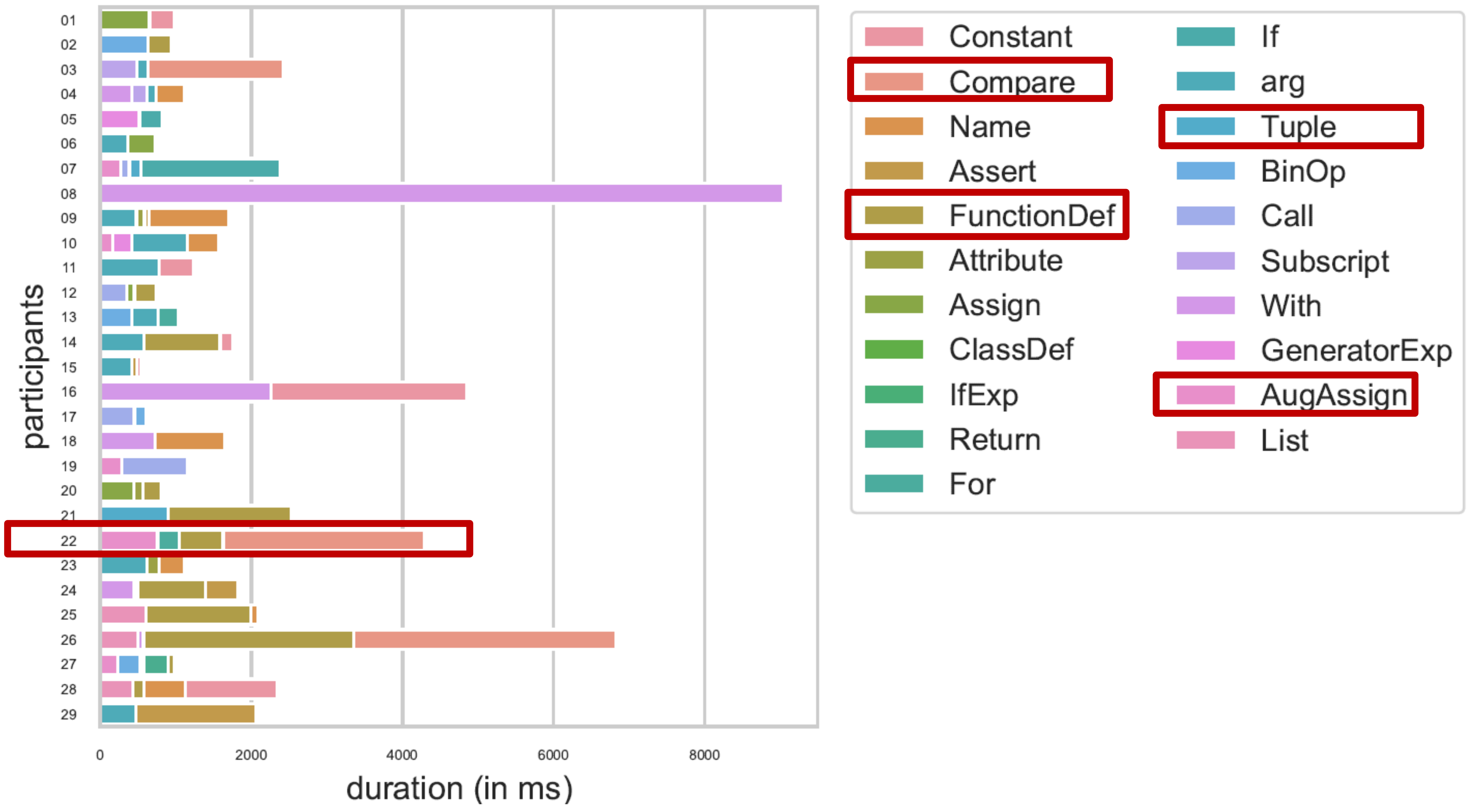
Not Confident: 0 Slightly Confident: 1 Moderately Confident: 2 Confident: 3 Very Confident: 4

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

```python
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))
```

Fixations of four participants for **functional** task classification

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



Gazes on all **mixed** task token types by all participants

Discussion

```
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
```

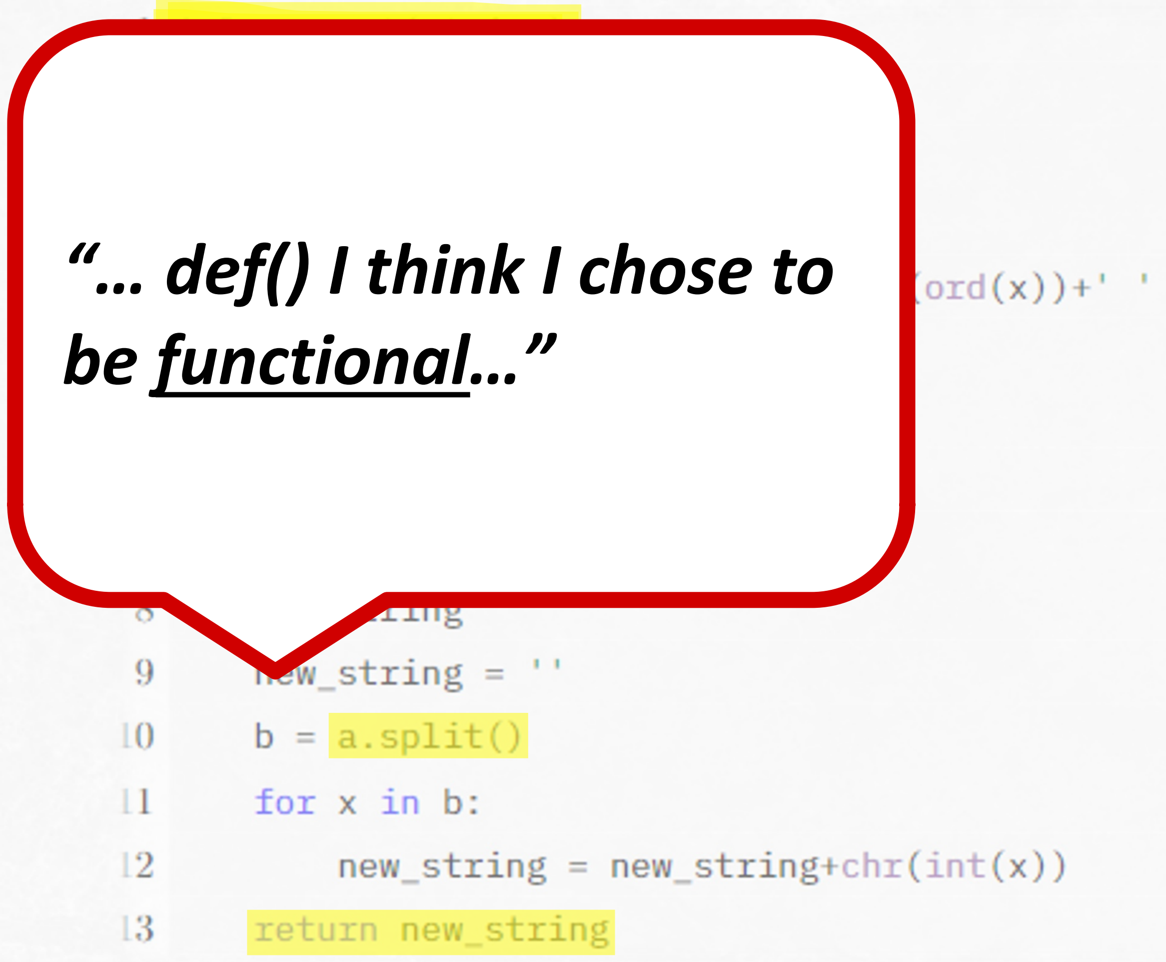
Incorrect judgement

```
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
```

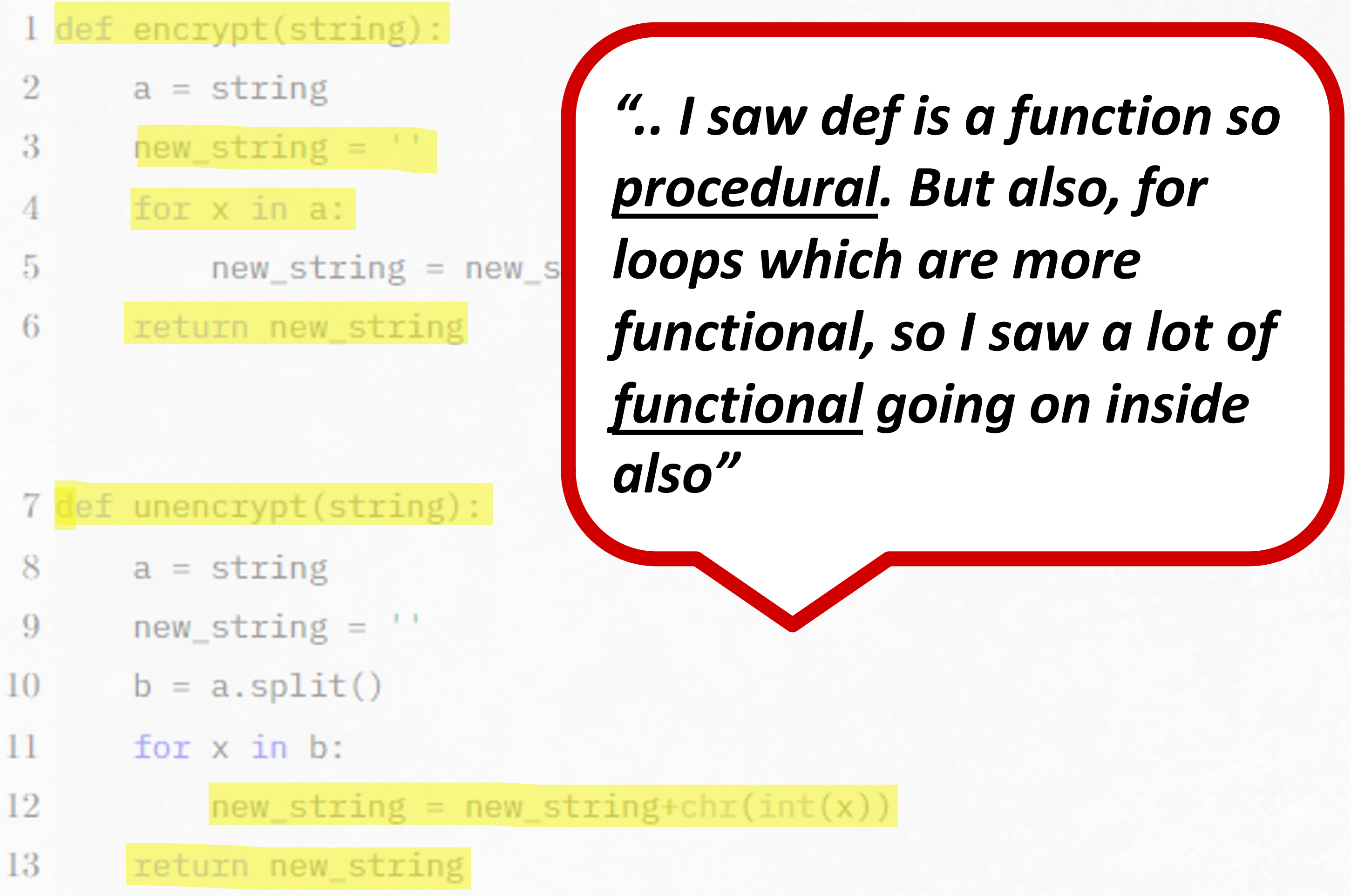
Correct judgement

Comparing fixations for **Procedural** paradigm (Task 1)

Discussion



Incorrect judgement

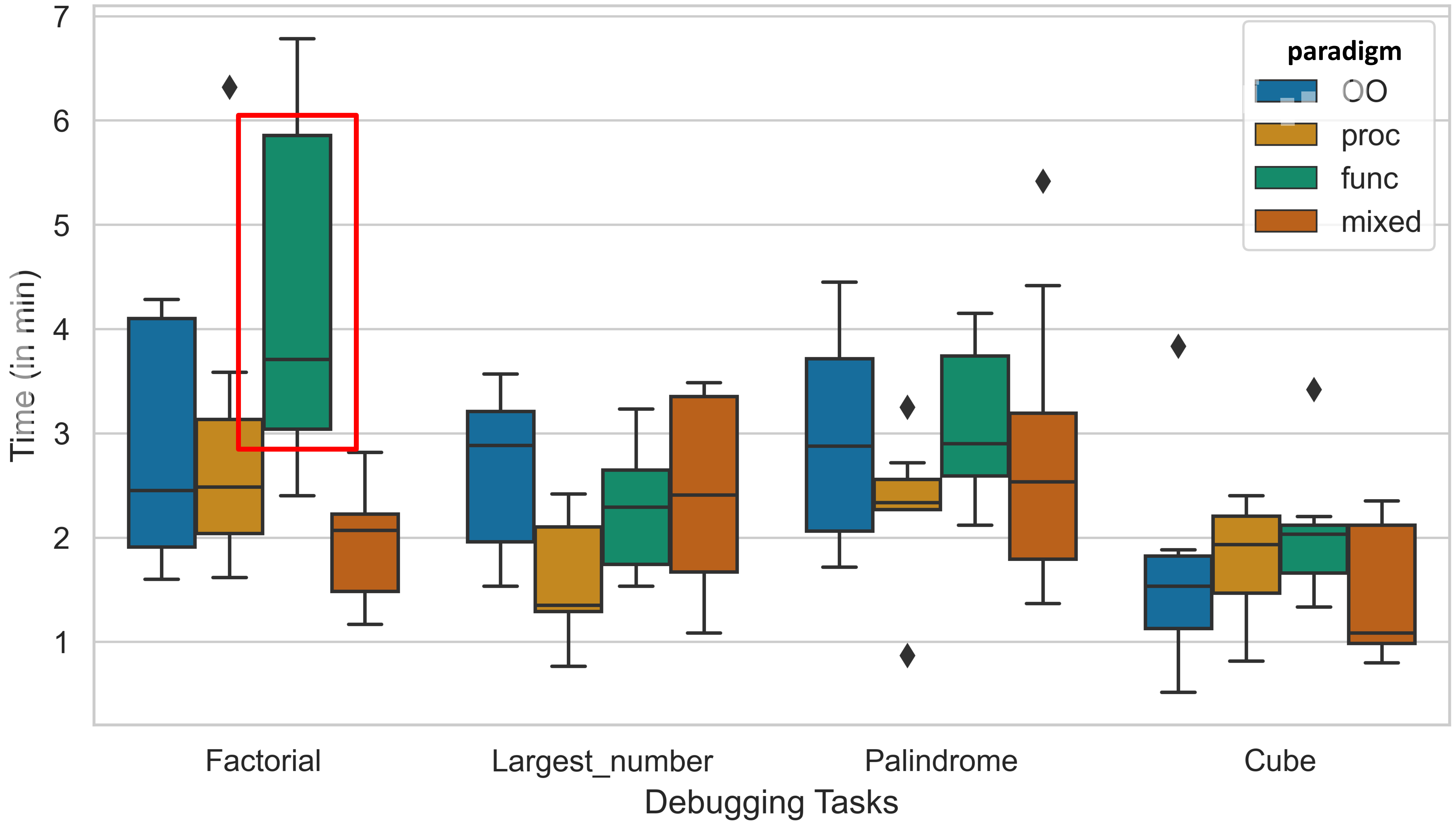


Correct judgement

Comparing fixations for Procedural paradigm (Task 1)

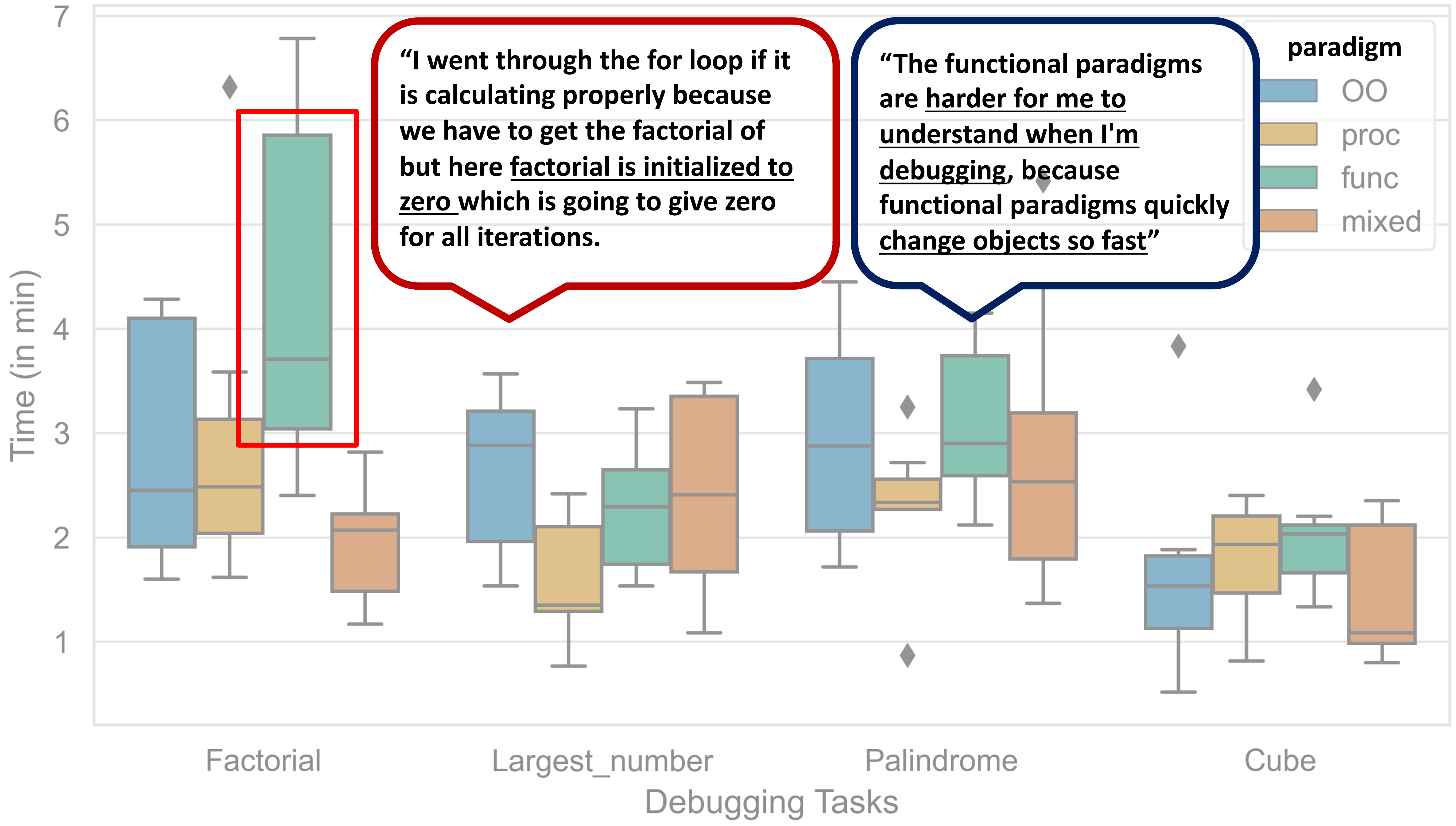
Results: Task Category 2 (Bug Localization)

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



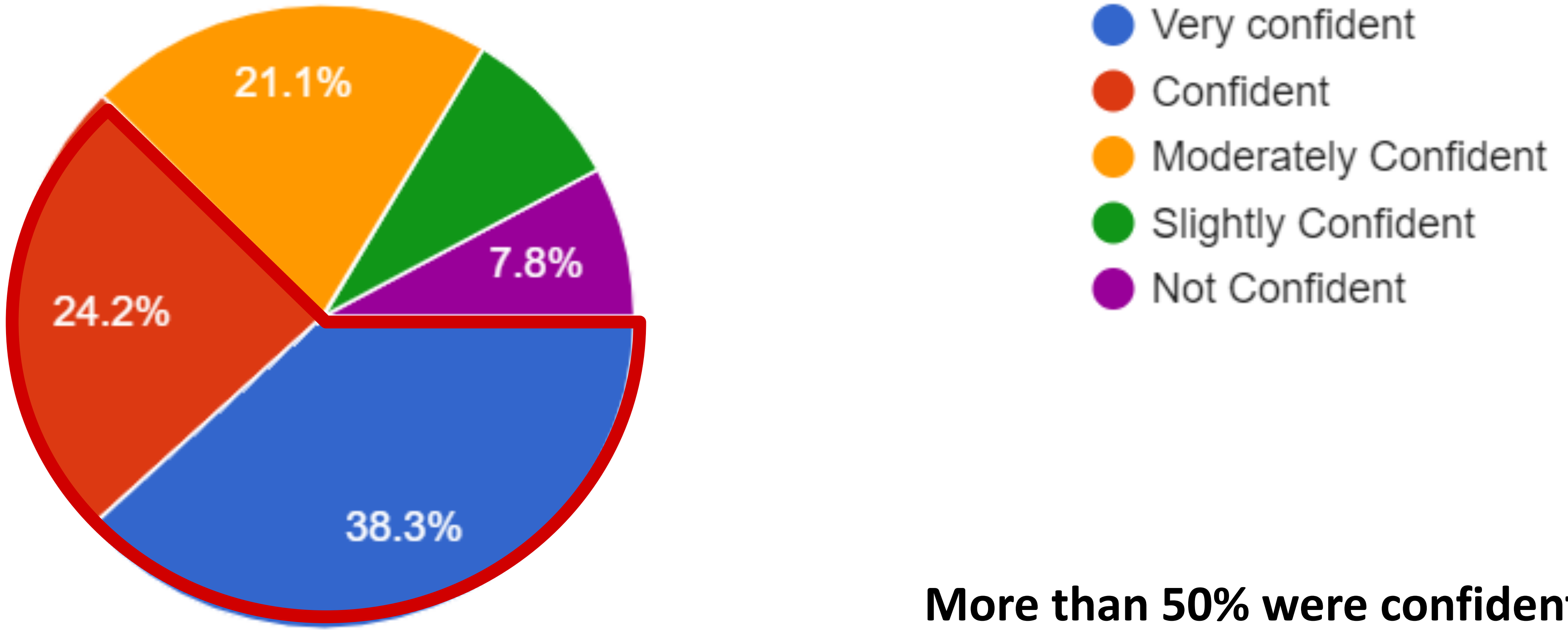
Time taken for Bug Localization **(Task Category 2)**

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



Time taken for Bug Localization **(Task Category 2)**

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



More than 50% were confident with debugging!

Confidence Levels for Logical Debugging (Task 2)

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

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

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

Effect of paradigm on correctness and debugging

Discussion

```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}')
```

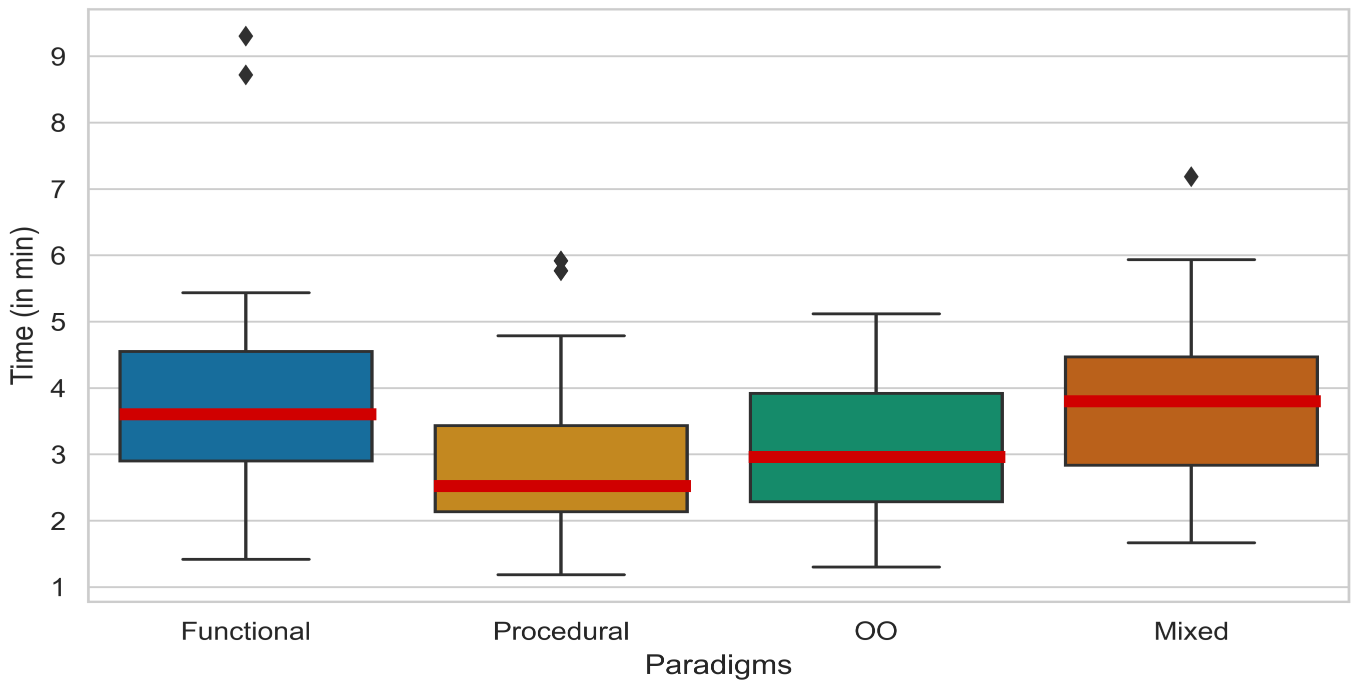
Correct Judgment

```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}')
```

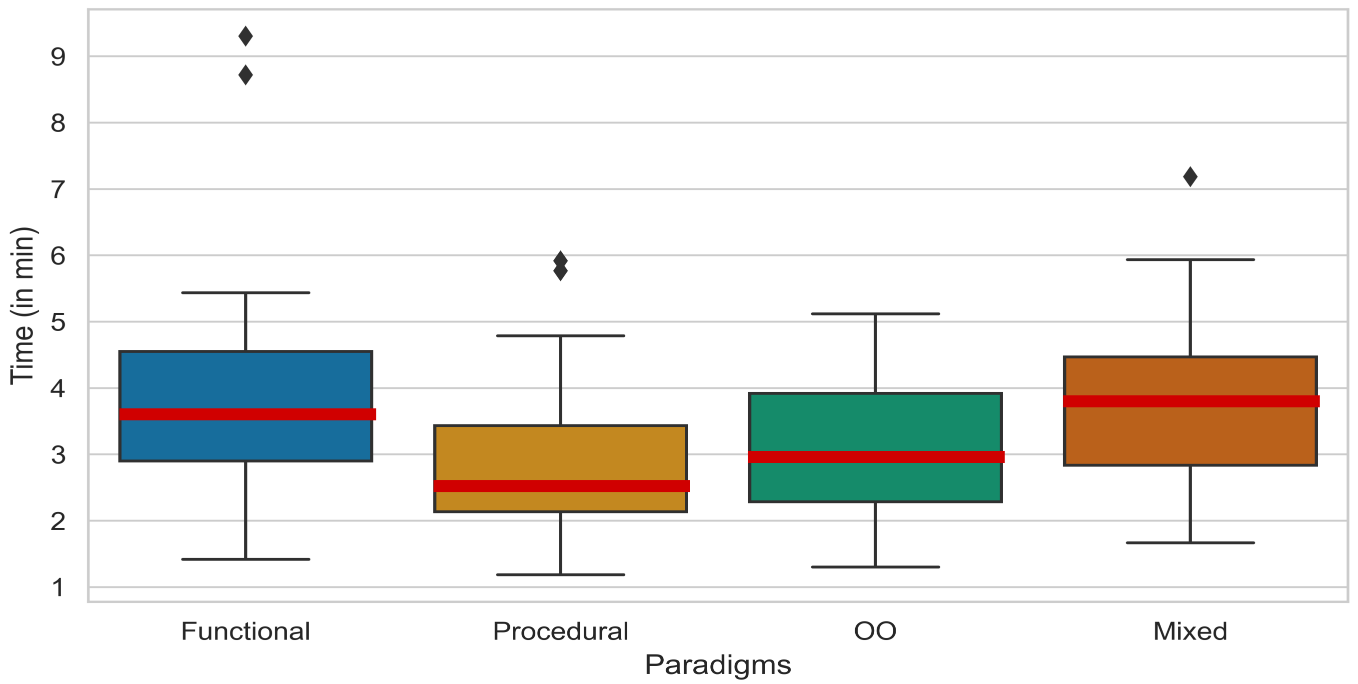
Incorrect Judgment

Comparing fixations for **Factorial mixed** paradigm (Task Category 2)

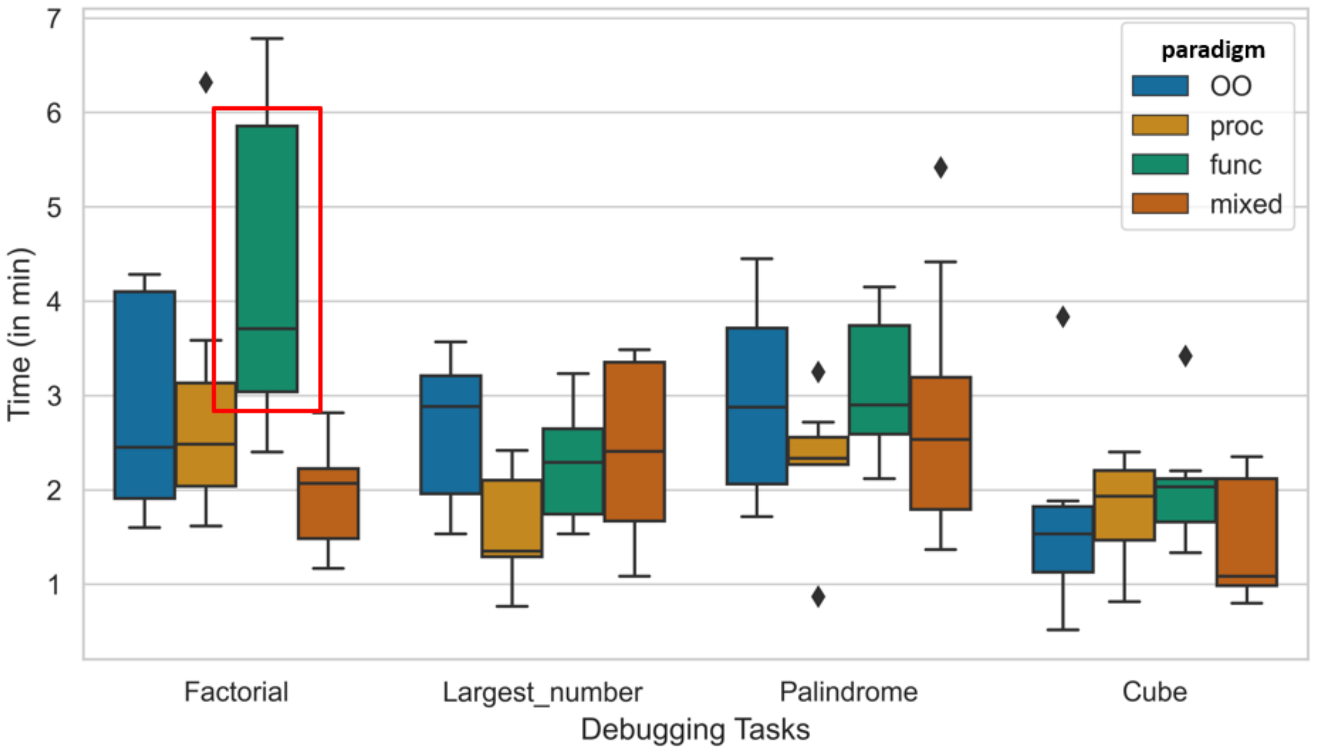
CONTRIBUTIONS
AND
CONCLUSION

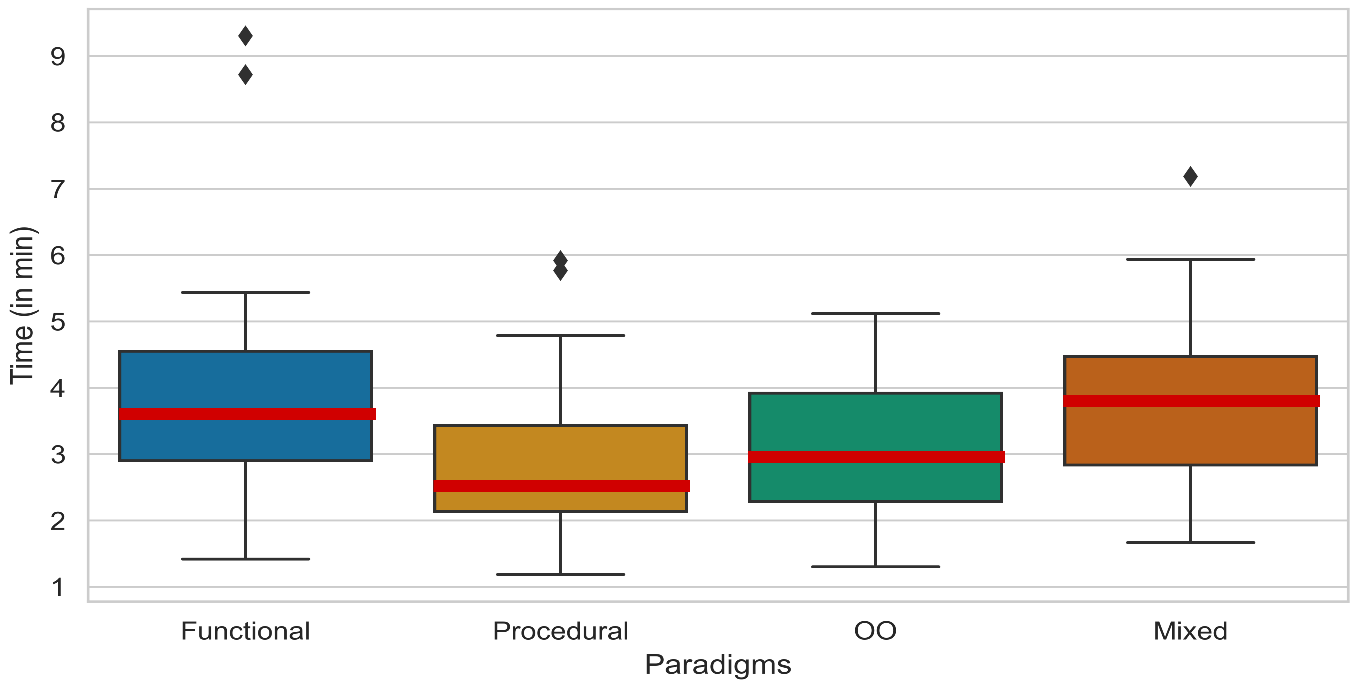


CONTRIBUTIONS AND CONCLUSION

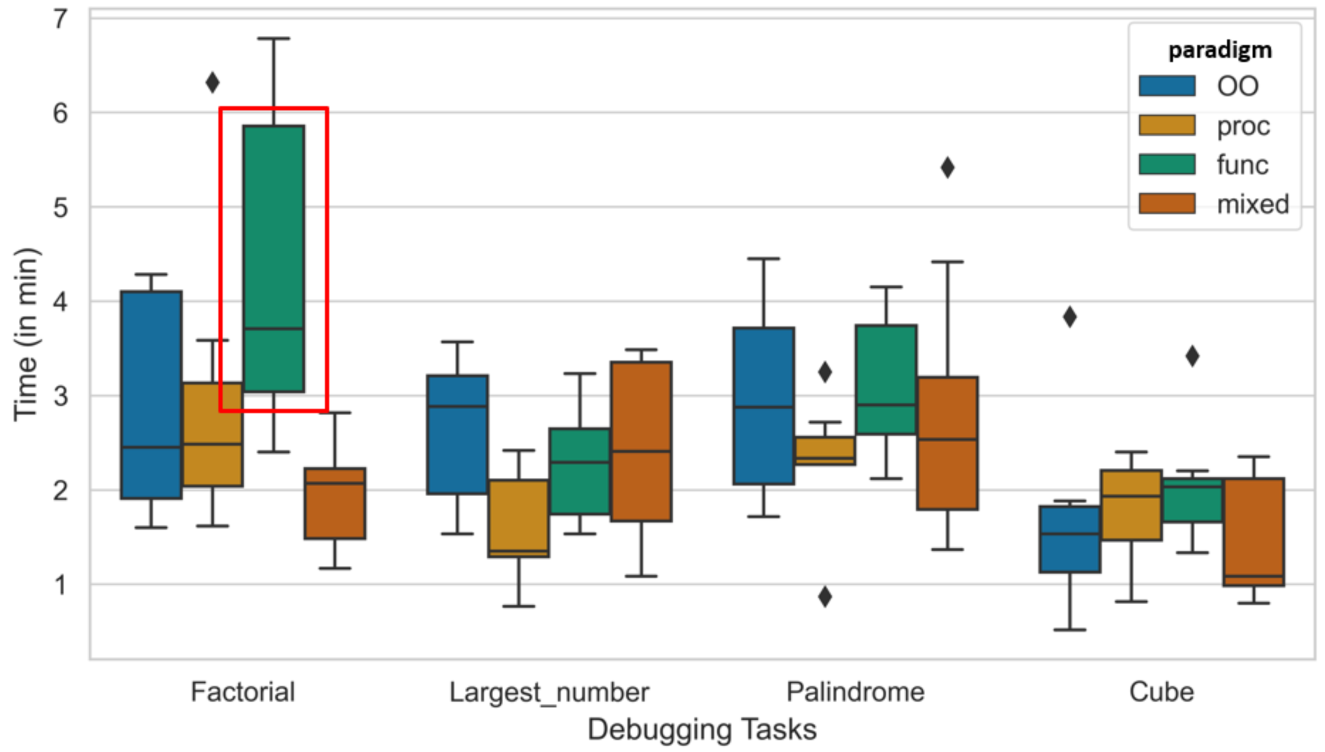


CONTRIBUTIONS AND CONCLUSION

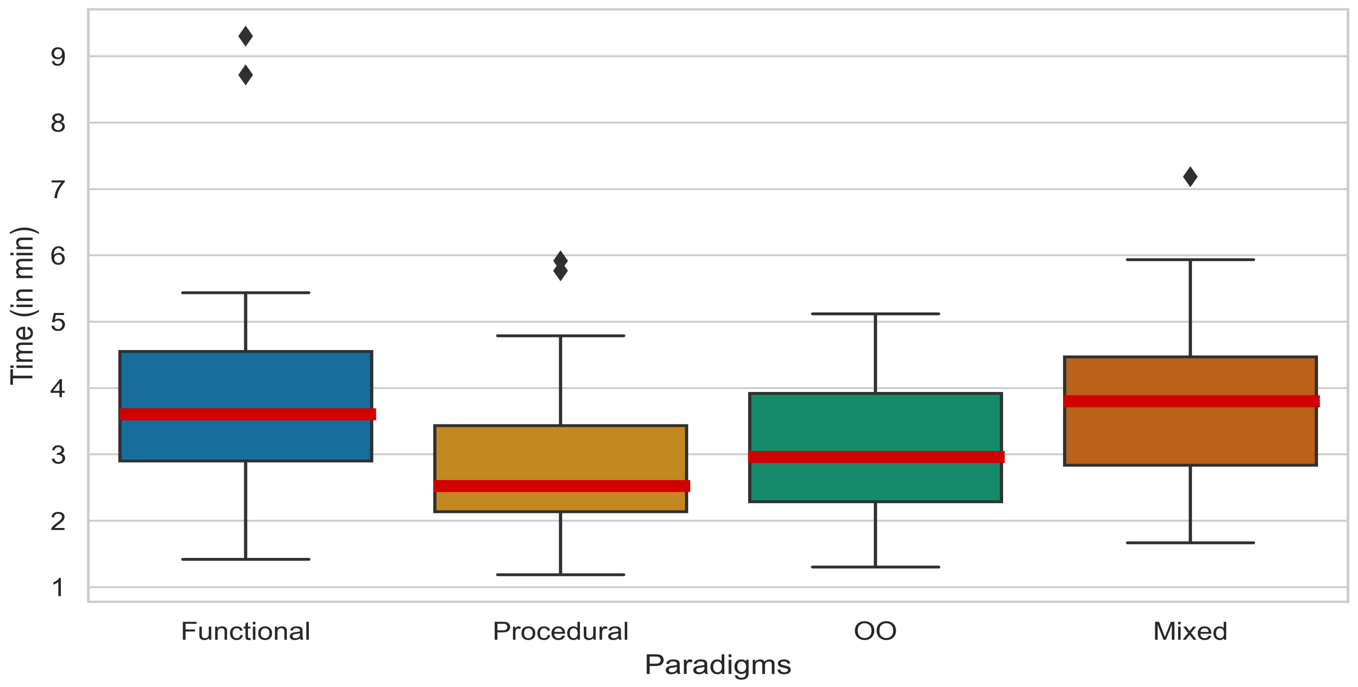




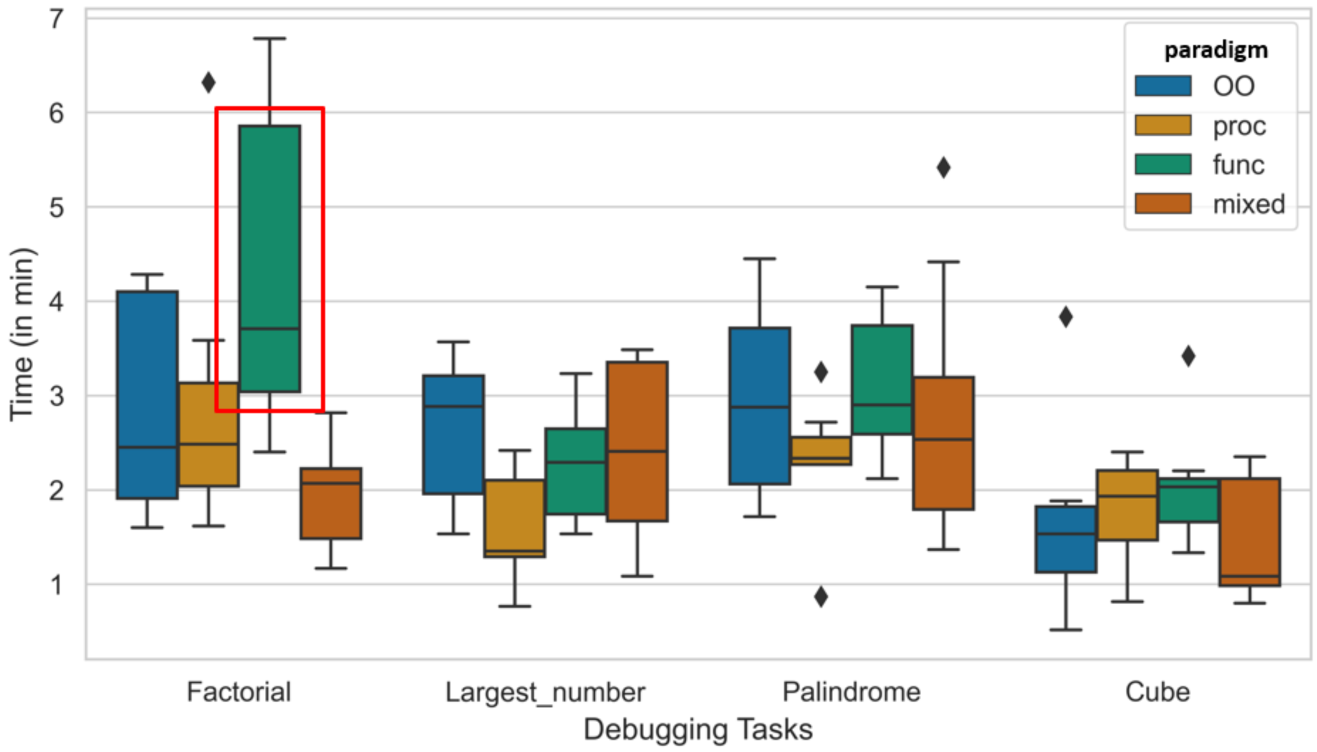
CONTRIBUTIONS AND CONCLUSION




```
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))
```



CONTRIBUTIONS AND CONCLUSION




```python
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))
```