

February 24, 2024



Department of Computer Science and Engineering  
Islamic University of Technology (IUT)  
A subsidiary organ of OIC

Software Maintenance Lab 03  
Azmayen Fayek Sabil, 190042122

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                        | <b>2</b>  |
| 1.1      | Tic Tac Toe Game . . . . .                 | 2         |
| 1.1.1    | Project Overview . . . . .                 | 2         |
| 1.1.2    | implementation . . . . .                   | 2         |
| <b>2</b> | <b>Loguru</b>                              | <b>3</b>  |
| 2.1      | Usage in the Project . . . . .             | 3         |
| 2.2      | Insight into the Output . . . . .          | 3         |
| 2.3      | Example for Maintenance Purposes . . . . . | 3         |
| 2.4      | Log Configuration Example . . . . .        | 3         |
| <b>3</b> | <b>Pysnooper</b>                           | <b>5</b>  |
| 3.1      | Usage in the Project . . . . .             | 5         |
| 3.2      | Insight into the Output . . . . .          | 5         |
| 3.3      | Example for Maintenance Purposes . . . . . | 5         |
| 3.4      | pysnooper Example . . . . .                | 5         |
| <b>4</b> | <b>Viztracer</b>                           | <b>6</b>  |
| 4.1      | Usage in the Project . . . . .             | 6         |
| 4.2      | Insight into the Output . . . . .          | 6         |
| 4.3      | Example for Maintenance Purposes . . . . . | 6         |
| 4.3.1    | viztracer Usage Example . . . . .          | 7         |
| <b>5</b> | <b>Snakeviz</b>                            | <b>8</b>  |
| 5.1      | Usage in the Project . . . . .             | 8         |
| 5.2      | Insight into the Output . . . . .          | 8         |
| 5.3      | Example for Maintenance Purposes . . . . . | 8         |
| 5.3.1    | snakeviz Usage Example . . . . .           | 8         |
| <b>6</b> | <b>Conclusion And Take away</b>            | <b>10</b> |
| 6.1      | Overview of Tools . . . . .                | 10        |
| 6.2      | Comparison . . . . .                       | 10        |
| 6.2.1    | Loguru . . . . .                           | 10        |
| 6.2.2    | pysnooper . . . . .                        | 10        |
| 6.2.3    | viztracer . . . . .                        | 10        |
| 6.2.4    | snakeviz . . . . .                         | 10        |
| 6.3      | Personal Preference . . . . .              | 10        |

# 1 Introduction

## 1.1 Tic Tac Toe Game

The Tic-Tac-Toe game project aims to implement a classic two-player game where players take turns marking cells in a 3x3 grid. The game is played by two opponents, one using 'X' and the other 'O', with the objective of forming a horizontal, vertical, or diagonal line of their respective symbols.

### 1.1.1 Project Overview

The project involves creating a console-based Tic-Tac-Toe game in Python. The implementation will include features such as displaying the game board, accepting user input for moves, checking for a winning condition, and handling invalid moves. The goal is to provide an enjoyable and interactive gaming experience.

### 1.1.2 implementation

The system architecture will consist of a main game loop, user input handling, game board representation, and result evaluation. The architecture aims for modularity and flexibility, allowing for future additions or modifications.

The game implementation will include Python functions for displaying the game board, accepting player moves, checking for a winning condition, and handling user input errors. Code modularity will be emphasized to improve readability and maintainability.

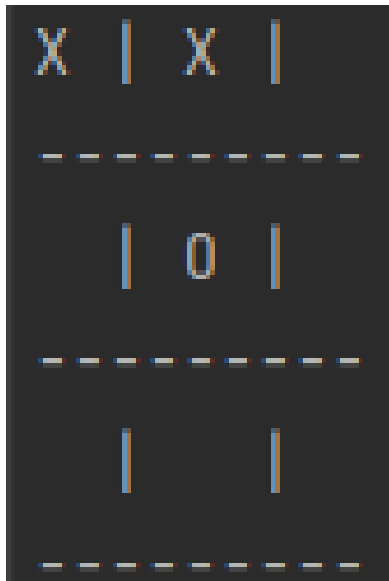


Figure 1: TicTacToe Game Board

## 2 Loguru

### 2.1 Usage in the Project

Loguru, a logging library for Python, was employed in our project to handle logging and provide informative outputs during various stages of development. It was integrated to enhance logging functionalities beyond the default Python logging module.

### 2.2 Insight into the Output

Loguru simplifies the logging process by offering an easy-to-use interface. It allows setting up different sinks (outputs), formatting log messages, and managing log levels effortlessly. This resulted in more readable and structured logs, aiding in debugging and monitoring the project's behavior. In our case we are returning different log information like if the game has finished and if finished who won? Then if there is an error, we are also storing that in the log file. This way it is giving us a way to track our output.

### 2.3 Example for Maintenance Purposes

During maintenance, Loguru facilitated quick identification of issues and monitoring of system events. For example, by configuring a log sink to save logs to a file, we could review historical logs to trace the origin of errors, helping in diagnosing and fixing issues efficiently.

### 2.4 Log Configuration Example

Below is a sample Loguru configuration code snippet used in our project:

```
from loguru import logger

# Configure Loguru to log messages at the "INFO" level and above
logger.add("tictactoe.log", rotation="10 MB", level="INFO")
-----

----
----
----

if check_winner(board, current_player):
    print_board(board)
    logger.info(f"Player {current_player} wins!")
    break
elif is_board_full(board):
    print_board(board)
    logger.info("It's a draw!")
    break
else:
    current_player = 'O' if current_player == 'X' else 'X'
-----

-----
-----
-----

try:
```

```

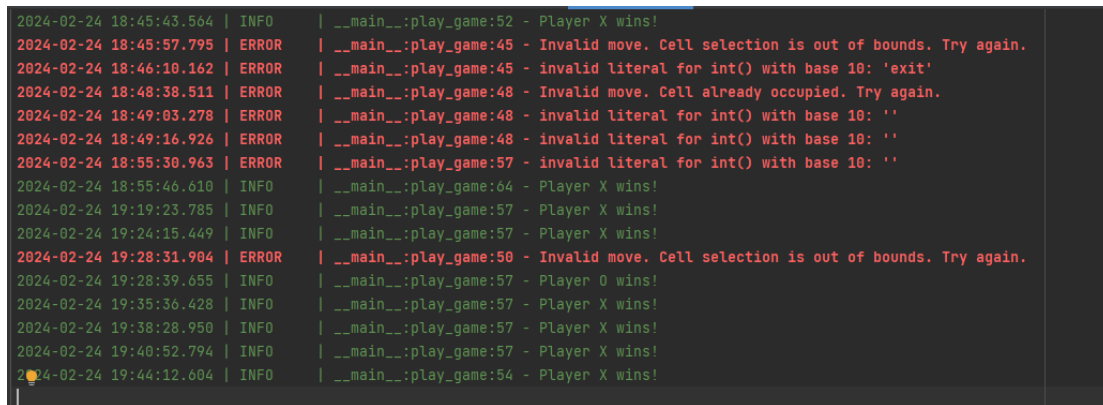
row = int(input(f"Player {current_player}, enter row (0-2): "))
col = int(input(f"Player {current_player}, enter column (0-2): "))

if not (0 <= row < 3 and 0 <= col < 3):
    raise ValueError("Invalid move. Cell selection is out of bounds. Try aga

if not is_valid_move(board, row, col):
    raise ValueError("Invalid move. Cell already occupied. Try again.")
except ValueError as e:
    logger.error(e)
    continue
-----
-----
-----

```

In this example, two log sinks are configured. One logs general information with a daily rotation, and the other logs errors with a weekly rotation. This setup aids in separating different log types and managing log files effectively.



```

2024-02-24 18:45:43.564 | INFO | __main__:play_game:52 - Player X wins!
2024-02-24 18:45:57.795 | ERROR | __main__:play_game:45 - Invalid move. Cell selection is out of bounds. Try again.
2024-02-24 18:46:10.162 | ERROR | __main__:play_game:45 - invalid literal for int() with base 10: 'exit'
2024-02-24 18:48:38.511 | ERROR | __main__:play_game:48 - Invalid move. Cell already occupied. Try again.
2024-02-24 18:49:03.278 | ERROR | __main__:play_game:48 - invalid literal for int() with base 10: ''
2024-02-24 18:49:16.926 | ERROR | __main__:play_game:48 - invalid literal for int() with base 10: ''
2024-02-24 18:55:30.963 | ERROR | __main__:play_game:57 - invalid literal for int() with base 10: ''
2024-02-24 18:55:46.610 | INFO | __main__:play_game:64 - Player X wins!
2024-02-24 19:19:23.785 | INFO | __main__:play_game:57 - Player X wins!
2024-02-24 19:24:15.449 | INFO | __main__:play_game:57 - Player X wins!
2024-02-24 19:28:31.904 | ERROR | __main__:play_game:50 - Invalid move. Cell selection is out of bounds. Try again.
2024-02-24 19:28:39.655 | INFO | __main__:play_game:57 - Player O wins!
2024-02-24 19:35:36.428 | INFO | __main__:play_game:57 - Player X wins!
2024-02-24 19:38:28.950 | INFO | __main__:play_game:57 - Player X wins!
2024-02-24 19:40:52.794 | INFO | __main__:play_game:57 - Player X wins!
2024-02-24 19:44:12.604 | INFO | __main__:play_game:54 - Player X wins!

```

Figure 2: Loguru Log File

## 3 Pysnooper

### 3.1 Usage in the Project

The project extensively utilized ‘pysnooper’, a Python debugging library, to gain valuable insights into the execution flow and variable values during specific code sections during the runtime. This tool was integrated as part of our interactive debugging strategy, playing a important role in identifying and troubleshooting issues effectively.

### 3.2 Insight into the Output

The use of ‘pysnooper’ significantly enhanced the debugging process by providing a comprehensive trace of code execution. Each executed step, along with corresponding variable values, was meticulously logged. This detailed output empowered developers to analyze the program’s flow, aiding in understanding the sequence of operations, identifying unexpected behavior, and enhancing overall code efficiency. In our case, this use of library is giving us runtime output of diffrenet variables that are used in the project. Thus helping us to keep track of the variable in each iteration.

### 3.3 Example for Maintenance Purposes

During maintenance tasks, ‘pysnooper’ emerged as a valuable asset for isolating and resolving issues promptly. By strategically placing the ‘@pysnooper.snoop()’ decorator on specific functions, we were able to observe the real-time behavior of these functions. This capability proved instrumental in swiftly locating and addressing bugs, contributing to efficient maintenance workflows.

### 3.4 pysnooper Example

The following code snippet illustrates a sample use of ‘pysnooper’ within our project:

```
from pysnooper import snoop

# Decorate the function with pysnooper
@snoop()
def example_function(x, y):
    result = x + y
    return result

# Function call with pysnooper tracing
example_function(10, 5)
```

In this example, the ‘@snoop()’ decorator is applied to the ‘example\_function’. When the function is invoked, ‘pysnooper’ dynamically generates a detailed output for each step of execution, providing valuable insights into the function’s behavior.

This is the output that I got after I ran the program and used pysnooper to listen the changes.

```
G:\190042122_tictactoe_game\Scripts\python.exe E:\University-4-2\SWE4802_Lab\Lab03\190042122_tictactoe_game\main.py
Source path:... E:\University-4-2\SWE4802_Lab\Lab03\190042122_tictactoe_game\main.py
00:02:56.134474 call      31 def play_game():
00:02:56.134474 line      32     board = [[' ' for _ in range(3)] for _ in range(3)]
New var:..... board = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
00:02:56.135475 line      33     current_player = 'X'
New var:..... current_player = 'X'
00:02:56.135475 line      35     while True:
00:02:56.135475 line      36         print_board(board)
00:02:56.135475 line      37         try:
00:02:56.135475 line      38             row = int(input(f"Player {current_player}, enter row (0-2): "))
| |
-----
| |
-----
| |
-----
Player X, enter row (0-2): 0
Player X, enter column (0-2): New var:..... row = 0
00:03:05.527090 line      39             col = int(input(f"Player {current_player}, enter column (0-2): "))
|
New var:..... col = 1
00:03:07.670667 line      41                 if not (0 <= row < 3 and 0 <= col < 3):
00:03:07.670667 line      44                 if not is_valid_move(board, row, col):
00:03:07.670667 line      50                 board[row][col] = current_player
Modified var:.. board = [[' ', 'X', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
00:03:07.670667 line      52                 if check_winner(board, current_player):
00:03:07.670667 line      56                 elif is_board_full(board):
00:03:07.670667 line      61                 current_player = 'O' if current_player == 'X' else 'X'
Modified var:.. current_player = 'O'
00:03:07.670667 line      35     while True:
00:03:07.670667 line      36         print_board(board)
00:03:07.670667 line      37         try:
00:03:07.670667 line      38             row = int(input(f"Player {current_player}, enter row (0-2): "))
| X |
-----
| |
-----
| |
-----
```

Figure 3: Pysnooper Log

## 4 Viztracer

### 4.1 Usage in the Project

‘viztracer’, a Python profiler and visualization tool, was employed in our project to analyze and visualize the performance of specific code sections. It served as a powerful tool for profiling and understanding the runtime behavior of the application.

### 4.2 Insight into the Output

‘viztracer’ provides detailed profiling information and generates interactive HTML reports. The output includes function call information, time taken by each function, and a visual representation of the code execution flow. This facilitated a deep understanding of the performance characteristics of the code. This visualization gives us a way of diving deep in the working of the code. We can get much closer to the behind the scene of the code, Thus helping us to optimizie and solve issues more efficiently.

### 4.3 Example for Maintenance Purposes

During maintenance, ‘viztracer’ played a vital role in identifying performance bottlenecks and optimizing critical sections of the code. By generating and analyzing HTML reports, developers could pinpoint areas for improvement, leading to more efficient and responsive code.

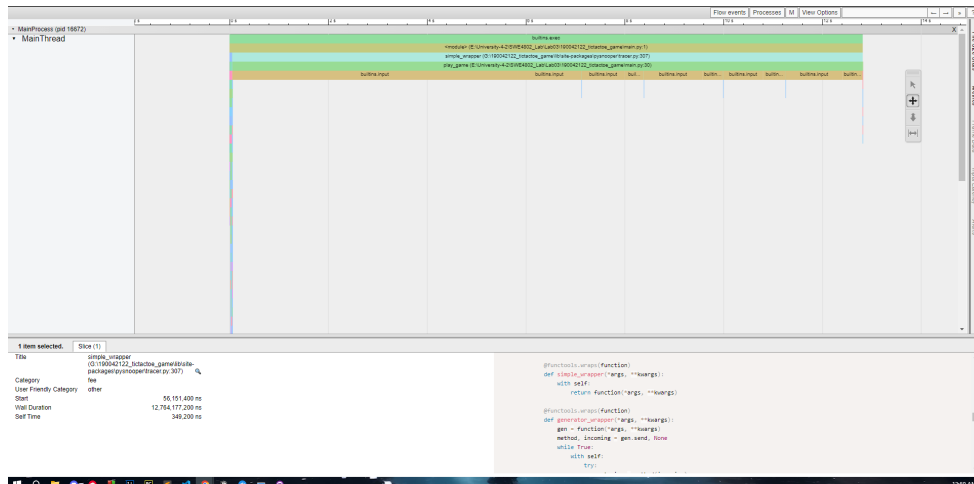


Figure 4: Viztracer Web UI

### 4.3.1 viztracer Usage Example

Below is a sample code snippet illustrating the use of ‘viztracer’ in our project:

```

from viztracer import VizTracer

# Create a VizTracer instance
tracer = VizTracer()

# Start tracing
tracer.start()

# Code section to be traced
# ...

# Stop tracing
tracer.stop()

# Generate HTML report
tracer.save("viztracer_report.html")

```

In this example, a ‘VizTracer’ instance is created, and the tracing is started and stopped around the specific code section to be analyzed. The generated HTML report provides a visual representation of the code execution flow and performance metrics.



## 5 Snakeviz

### 5.1 Usage in the Project

‘snakeviz’, a Python profiling tool with a web-based interface, was integrated into our project for profiling and visualizing the performance of specific code sections. It served as an effective tool for identifying performance bottlenecks and optimizing critical parts of the code.

### 5.2 Insight into the Output

‘snakeviz’ generates interactive visualizations in the form of sunburst charts, flame graphs, and tables. These visualizations provide insights into the time consumption of functions, helping developers identify functions that contribute most to the overall runtime. This graphical representation aids in making informed optimization decisions.

### 5.3 Example for Maintenance Purposes

During maintenance, ‘snakeviz’ proved valuable in analyzing and optimizing code performance. By profiling specific functions or code blocks, developers could identify areas of improvement and focus efforts on optimizing the most time-consuming parts of the codebase.

#### 5.3.1 snakeviz Usage Example

Below is a sample code snippet illustrating the use of ‘snakeviz’ in our project:

```
# Install snakeviz (if not installed)
# pip install snakeviz

# TO CREATE PROFILE USE THIS IN COMMAND LINE
python -m cProfile -o snakeviz_log.profile main.py

# TO SEE THE VISUALIZATION
snakeviz snakeviz_log.profile
```

In this example, the ‘snakeviz’ command is used to profile a Python script (‘my\_script.py’). This command generates a visualization and opens it in a web browser, allowing developers to interactively explore the profiling results.

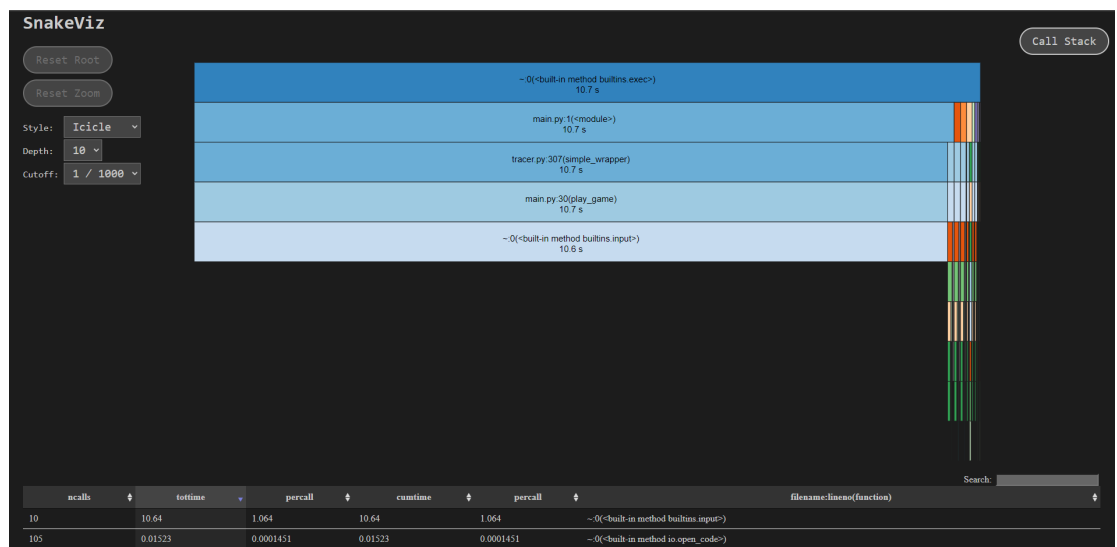


Figure 5: SnakeViz Visualisation

|           | calls | ↑ | tottime   | ↓ | percall   | ↑ | cumtime   | ↑ | percall   | ↓ | filename:lineno(function)                              | ↑ |
|-----------|-------|---|-----------|---|-----------|---|-----------|---|-----------|---|--|---|
| 10        |       |   | 10.64     |   | 1.064     |   | 10.64     |   | 1.064     |   | ~O(<built-in method builtins.input>)                   |   |
| 105       |       |   | 0.01523   |   | 0.0001451 |   | 0.01523   |   | 0.0001451 |   | ~O(<built-in method io.open_code>)                     |   |
| 550       |       |   | 0.01259   |   | 2.29e-05  |   | 0.01259   |   | 2.29e-05  |   | ~O(<built-in method nt.start>)                         |   |
| 1         |       |   | 0.0078    |   | 0.0078    |   | 10.65     |   | 10.65     |   | main.py:30(play_game)                                  |   |
| 9         |       |   | 0.006758  |   | 0.0007509 |   | 0.008554  |   | 0.0009504 |   | ~O(<built-in method _imp.create_dynamic>)              |   |
| 105       |       |   | 0.001968  |   | 5.684e-05 |   | 0.001968  |   | 5.684e-05 |   | ~O(<built-in method marshal.loads>)                    |   |
| 1428      |       |   | 0.004959  |   | 3.473e-06 |   | 0.00759   |   | 5.315e-06 |   | <frozen importlib._bootstrap_external>:96(path_join)   |   |
| 105       |       |   | 0.002887  |   | 2.75e-05  |   | 0.002887  |   | 2.75e-05  |   | ~O(<a method 'read' of 'io.BufferedReader' objects>)   |   |
| 322/320   |       |   | 0.002729  |   | 8.527e-06 |   | 0.007643  |   | 2.388e-05 |   | ~O(<built-in method builtins._build_class_>)           |   |
| 36        |       |   | 0.001966  |   | 5.462e-05 |   | 0.001966  |   | 5.462e-05 |   | ~O(<built-in method builtins.print>)                   |   |
| 289       |       |   | 0.001684  |   | 5.826e-06 |   | 0.01939   |   | 6.709e-05 |   | <frozen importlib._bootstrap_external>:1527(find_spec) |   |
| 393       |       |   | 0.001175  |   | 2.99e-06  |   | 0.002185  |   | 5.56e-06  |   | ntpath.py:463(normpath)                                |   |
| 113/22    |       |   | 0.001046  |   | 4.755e-05 |   | 0.0028    |   | 0.0001273 |   | src_parse.py:493(parse)                                |   |
| 105       |       |   | 0.00098   |   | 9.333e-06 |   | 0.00098   |   | 9.333e-06 |   | ~O(<a method '__exit__' of 'io.IOBase' objects>)       |   |
| 6314      |       |   | 0.0009544 |   | 1.512e-07 |   | 0.0009544 |   | 1.512e-07 |   | ~O(<a method 'startswith' of 'str' objects>)           |   |
| 9         |       |   | 0.0009069 |   | 0.0001008 |   | 0.0009069 |   | 0.0001008 |   | ~O(<built-in method _imp.exec_dynamic>)                |   |
| 134       |       |   | 0.0008039 |   | 5.999e-06 |   | 0.02275   |   | 0.0001698 |   | <frozen importlib._bootstrap>:921(_find_spec)          |   |
| 105       |       |   | 0.0007981 |   | 7.601e-06 |   | 0.0315    |   | 0.0003    |   | <frozen importlib._bootstrap_external>:950(get_code)   |   |
| 18        |       |   | 0.0007264 |   | 4.036e-05 |   | 0.002459  |   | 0.0001366 |   | enum.py:180(_new_)                                     |   |
| 1428      |       |   | 0.0007037 |   | 4.928e-07 |   | 0.0009206 |   | 6.447e-07 |   | <frozen importlib._bootstrap_external>:119(<listcomp>) |   |
| 299       |       |   | 0.0006443 |   | 2.155e-06 |   | 0.0006783 |   | 2.269e-06 |   | enum.py:470(_setup_)                                   |   |
| 8703/8501 |       |   | 0.0006306 |   | 7.418e-08 |   | 0.0006683 |   | 7.861e-08 |   | ~O(<built-in method builtins.len>)                     |   |
| 188/21    |       |   | 0.0006138 |   | 3.832e-05 |   | 0.001687  |   | 8.008e-05 |   | src_compile.py:71(compile)                             |   |

Figure 6: SnakeViz Visualisation

## 6 Conclusion And Take away

### 6.1 Overview of Tools

In our exploration of profiling and visualization tools, we have discussed Loguru for logging, pynsnoop for real-time tracing, viztracer for runtime profiling, and snakeviz for visualizing profiling results. Each tool serves distinct purposes and offers unique features.

### 6.2 Comparison

#### 6.2.1 Loguru

Loguru works as a versatile logging library, providing easy configuration, structured output, and efficient log handling. It is suitable for capturing various log levels and organizing logs for debugging and monitoring purposes.

#### 6.2.2 pynsnoop

pynsnoop stands out as a runtime tracing tool, offering real-time insights into code execution and variable values. It aids in interactive debugging, making it valuable for identifying issues during development and maintenance.

#### 6.2.3 viztracer

viztracer is a comprehensive runtime profiler, generating detailed reports and visualizations. Its interactive HTML reports and profiling capabilities make it a robust choice for understanding code performance and optimizing critical sections.

#### 6.2.4 snakeviz

snakeviz provides a web-based interface for visualizing profiling results, offering sunburst charts and flame graphs. It excels in presenting a graphical representation of time consumption in functions, aiding in the identification of performance bottlenecks.

### 6.3 Personal Preference

In my working software projects, the choice of tools would depend on the specific requirements and nature of the project. For logging and general debugging, Loguru's simplicity and flexibility make it a preferred choice. For real-time tracing during development, pynsnoop's ease of use and interactive debugging features are valuable. For comprehensive runtime profiling and visualizations, viztracer and snakeviz provide in-depth insights and clear graphical representations, making them suitable for optimizing code performance.