Usability Challenges of Profiling Tools

A Project for CSE 291: Programmers are People Too

Samir Rashid [1], Amber Olsen [1] and Kyle Trinh [1]

[1] *University of California, San Diego, California*

Abstract

Profilers explain performance of programs. We conducted an exploratory study of students new to using profilers to find usability issues with existing tools. We find challenges understanding profiler output and finding relevant profiler information. We propose pytrail, a novel tool, which surfaces profiling data in the IDE and compare it to existing tools. We describe a collection of opportunities for improvement of profiling tools for inexperienced users.

Keywords: Profiling. Python. Usability of profiling tools. Empirical studies of programmers.

1 Introduction

Optimizing code during runtime is an essential part of programming. Profiling tools can be unwieldy and confusing to use. We find no prior art examining the usability of profiling tools[1]. Runtime efficiency is often an afterthought or requires extensive upfront planning. We suspect that 100% of programs could be improved by having faster runtime. If programmers could intuitively reason about runtime during development, they could save time by only optimizing important hotspots during the development process.

In our study, we compare cProfile[2], the standard command-line Python profiler, to pytrail, our custom, line-by-line in-IDE profiler. We sought to discover if having line-by-line information about the efficiency of code would help programmers or if they felt like the output of cProfile was sufficient. We conducted a within-user study to find which tool users preferred when profiling a small Python program. Although we were optimistic that pytrail would prove helpful to programmers, our study found mixed results. While not dissuading us of the value of in-IDE runtime feedback, we believe that bugs in pytrail led to user frustration with the tool. We hope that with those bugs being fixed, further research could show the usefulness of pytrail and in-IDE feedback.

RQ: Does integrating a Python profiling tool into an IDE help Python programmers of varied experience to detect unoptimized code more efficiently? In essence, this question can be subdivided into two parts: does in-IDE profiling data help users write more efficient code generally, and does pytrail specifically help to this end?

In order to answer these questions, we operated under the assumption that having profiler data readily available could help programmers of varied experience to improve their efficiency. With this assumption in mind, we sought to discover if gaining the line-by-line in-IDE feedback that pytrail could deliver would result in greater awareness of efficiency in users or greater user-satisfaction.

We decided to research this question because we found that while IntelliJ has a profiler built into the IDE for Java, this is not available for Python. We noticed that the Python tool pprofile did contain line-by-line code runtime data via the command line. We decided to create a wrapper for pprofile that we named pytrail and compare it to cProfile, the native Python command-line profiler.

cProfile is a sampling profiler built into Python. Currently programmers get no feedback on the runtime of sections of their program unless they instrument their code with profiling function calls and interpret the terse stack trace results. Our proposed tool sought to meet this gap in the development experience. This believed this would make it easier to diagnose slow runtime issues due to unoptimized code in the IDE.

Our Python profiler has line-by-line feedback to enable more granular performance analysis. We sought to discover if this could help programmers progressively evaluate and improve runtime issues.

1 https://scholar.google.com/scholar?hl=en&as_sdt=0,5&q=profiling+tools+usability
2 https://docs.python.org/3/library/profile.html

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

Organizers:
Michael Coblenz

This work is licensed under a Creative Commons Attribution 4.0 International License.

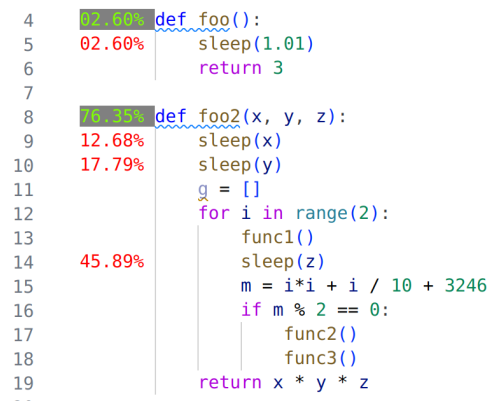

Figure 1. Excerpt of a task. Red annotations denote slow lines and what percent of runtime they take. Function definitions have highlighted cumulative runtime for entire function.

We believe that having more information about performance could have a positive impact on anyone who programs in Python. We think this is an issue of interest to anyone who programs in Python and wants to be efficient.

Overall, we had mixed results on the effectiveness of pytrail compared to cProfile, but we attribute this to bugs in pytrail as well as shortcomings of our study and not that by-line runtime feedback in an IDE is without value. We believe that line-by-line in-IDE feedback on runtime could be of benefit to all programmers, programming in whatever language. But, future studies would have to be performed to confirm this hypothesis.

2 Method

To recruit participants, we used Piazza (a forum built for educational discussion) and offered to enter participants in a drawing for $10. We devised two tasks labeled as task1.py and task2.py. Participants were first given either cProfile or pytrail and asked to analyze the runtime of task1. Then participants were given either pytrail or cProfile (whichever tool they did not receive for task1) and were asked to analyze the runtime of task2. At the end, participants were asked to fill out a Google form, with questions about the runtime of tasks (i.e. which line of code was the slowest? which function was the slowest?) which they filled out while reading the code. Users were given a maximum of 10 minutes to analyze the code for each task. On completion of analyzing both tasks, we gave them another Google form in which we asked them questions about how they felt about each tool, its ease of use, what issues they had, and what they would improve. We used their answers to the surveys listed above to analyze the effectiveness of each tool.

3 Results

We had a total of 5 participants in our study. They were all students pursuing bachelor's or master's degrees in Computer Science. With the allotted time of 10 minutes to complete each task, all participants completed tasks in less than the available time. Although we received after-study survey results for all 5 participants, we only have in-study assessments for 4 participants. Also, for one study participant we had trouble getting task1 to execute, so they did task2 for both cProfile and pytrail. Since we only have in-study assessments for 4 participants and one participant did task2 twice, we only had 3 legitimate answers to in-study questions about task1. Of those, 2 participants used pytrail for this task and 1 participant used cProfile. Participants differed on which line of code was the slowest in task1, one said line 29, one line 8, and finally one said line 5. Likewise for slowest function, we received more than one answers, 2 said main and 1 said foo2.

For task2, of the 4 participants for whom we have in-study assessments, half began with pytrail and the other half with cProfile. Again, we had varied responses to the slowest line of code. Two participants said that line 31 was the slowest, one said line 6, and one said line 32. For the slowest function, three participants said plu decomposition and one said recalibrate.

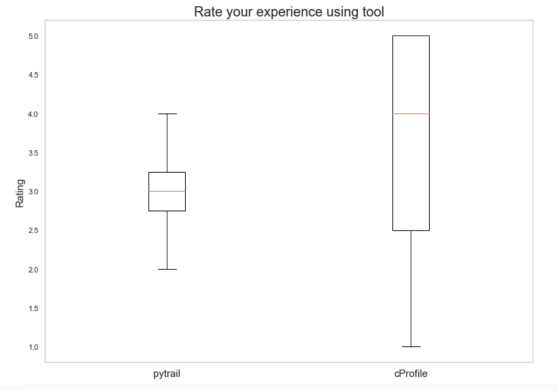

Figure 2. Users' overall rating of pytrail and cProfile

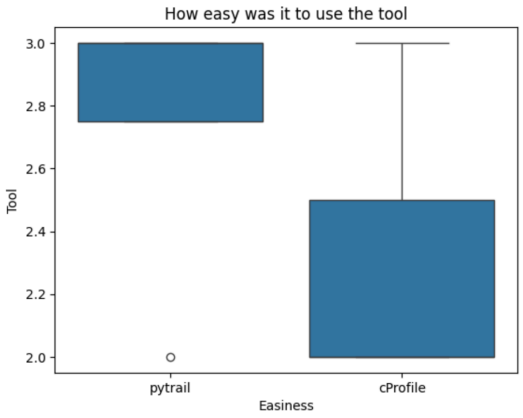

Figure 3. Users' rating of how easy it was to use pytrail and cProfile

For the after-study survey, three participants failed to answer some questions. As such, some of our results are not for the full five participants, but which participant failed to answer a given question varies. On a scale of 1-5 (5 being the best), we asked participants to rank their experience with each tool. One participant gave pytrail a score of 4, two participants gave it a score of 3, and one participant gave it a score of 2. This can be compared to how participants ranked their experience using cProfile where two participants gave it a score of 5, one gave it a score of 3 and one gave it a score of 1. (see Figure 2)

When asked about the difficulty of using the tools (where 5 is very easy, and 1 is difficult), three users gave pytrail a 3, and one user gave it a 2. For cProfile, we only had three responses to this question. One user gave it a 3 and two users gave it a 2. (see Figure 3)

Users gave feedback on issues they had with the tools. For pytrail, one user said they had no issues, another said that it ran fine but had confusing error messages. One didn't understand what the percentages meant. For cProfile, two users said they had no issues, one user complained that there were too many lines to go through.

They were then asked how they would improve the tools. For pytrail, users wanted clarity about the output; they didn't understand the percentages, and they wanted more instructions. For cProfile, one user requested that they break up the output visually and another user wanted descriptive information about the output.

Finally, participants were asked whether analyzing the code was easy or difficult. Overall, participants did not think the analysis was difficult, but one said they were unsure of their answers, and another said that they would have known the in-study runtime answers even without a profiler. They wrote, "It was relatively easy since there were parts that I would expect to take longer even if I didn't have a profiler to tell me so".

Our original research question was does integrating a Python profiling tool into an IDE help Python programmers of varied experience to detect unoptimized code more efficiently? Answering

that question specifically, we would have to answer no, we did not measure what we wanted to. Results as to whether having line-by-line in-IDE feedback on runtime was helpful were inconclusive. However, we believe this may be due to bugs with pytrail or shortcomings with our study itself.

3.1 Generalizability

Internal validity: Both tasks involved the participants reading Python code. We minimize the effect of prior knowledge by using a high-level language and by making the tasks short. We provided a development environment with all the tools set up for users and only provided guidance on running the tools. We used VSCode, which is a popular editor among students. We gave participants the same two tasks and switched the order to minimize any differing effects of the tasks. The short task length and participants only reading code limited the variability of results. Our study did not make participants edit code, so our results were specific to using the tools and how successful participants were in identifying runtime issues.

External validity: Participants did not edit code, which is unrealistic in a real coding situation. Further work could be done to increase the scope of the study to limit this effect. Our study does not find statistically significant results due to our small sample size (n=5). We treat our results as exploratory towards improving pytrail and preparing for a larger study.

Does having in-IDE line-by-line runtime feedback help programmers programming in any language? We suspect this is the case, but cannot use the results of our study to verify this. Bugs in pytrail coupled with shortcomings of the study itself make it difficult to draw conclusive evidence. However, the concept of using visualizations to surface tracing information in-IDE can be applied to any language. Future tools can leverage existing profilers and repurpose the pytrail interface.

4 Limitations

Some limitations of this study were that since our tool pytrail was newly developed, we did not have time to resolve all the issues with its use before releasing it to participants. As a result, there were still bugs that detracted from the benefit we hoped users would derive. Another problem is, we believe that for some of our functions, participants were able to ascertain which lines of code were the slowest without relying on any profiler. As a result, profiler output was extraneous to filling out our assessment, rendering it un-useful. Some of our participant data was incomplete and we had few participants in our study. One participant did task2 with both pytrail and cProfile, and another participant didn't do the in-study assessment and used pytrail with both tasks. These issues were all threats to the validity of the study, making it hard to determine the usefulness of these tools.

5 Discussion

5.1 User study process

We began work on this project with a lot of enthusiasm around our mission. We believed in the usefulness of in-IDE profiler data to help programmers write more efficient code. We were excited when the first prototype for pytrail was released by our team member, Samir Rashid, for perusal. Unfortunately, from the gate we had some issues with portability; team members Kyle Trinh and Amber Olsen struggled with getting the profiler to work on our devices. Although we resolved those initial issues, new issues arose. By the time we needed to release the tool to participants (because time had run short) there were still some bugs with pytrail yet to be resolved. We remain optimistic about the potential of this tool and the general usefulness of in-IDE profiler data to write better code. We can't say we were surprised by the mixed results we got, but we believe that the limited time was a factor in the some of the shortcomings in pytrail that otherwise could result in this tool performing better and receiving more positive feedback from users.

As we did our first user study, we found several issues with our study. P1 had many difficulties with running our study and answering questions. P1 had trouble needing to read through the code and command line output while also needing to refer to our survey questions for what information P1 needed to find. We had to modify the survey questions to be more clear. As runtime is hierarchical, some participants were confused by the runtime of different functions when everything is called from

main. However, pytrail measures time spent in the function and in external library calls but excludes time spent in calls to other functions in the source code. This is a fundamental issue with the tool and cannot be fixed as the alternative would show the 'main' function taking the entire runtime of a program.

5.2 Opportunities for tools

Based on our results, we compile a list of improvements for current profiling tools. Users found cProfile unintuitive and had to scroll through hundreds of lines of text. cProfile could be language aware and hide irrelevant information about internal runtime data. There are many Python tools that profile code. However, their usability is limited to command line interfaces with no visual feedback. We can generalize the results to see if coding tools with instant feedback benefit users more than a "manual" CLI code review. This could help novice as well as experienced Python programmers to make improvements to their code without hard mental operations.

6 Related Work

pytrail is built upon the Scalene [1] profiler. Scalene is a new profiler for Python and significantly improves upon the state of the art for profiler user-experience and accuracy. We use Scalene as a sampling profiler to minimize overhead and maximize the utility of pytrail.

7 Future Work

If we were to repeat this experiment again (or rather, for other individuals who wish to try to reproduce our results), we would have added more functions to the tool so that users could clearly see when pytrail finished annotating the code. Of course, with time limitations and a relatively new technology, such features were hindered by the time frame. Beyond our tool, we believe that if given the resources, future experiments could incorporate a pre-survey to screen the participants before testing them to ensure a representative sample size. The downside to more participants is the cost of running every experiment; we propose that future experiments should be run of GitHub Codespaces to reduce the number of hours spent on testing participants. The only downside is that it places significant trust onto the participants to not exceed the time limit.

In terms of task design, we initially had more tasks but found it to be relatively redundant; our tool was built with ease-of-use in mind, so more tasks would not have given more meaningful data. However, we suggest increasing the complexity of tasks to obfuscate API calls, library functions, etc. to mimic real-world applications.

Further research questions that build upon our work include:

1. Can integrating profiling tools into the IDE help programmers unfamiliar with profiling tools write more efficient code?
2. What challenges do programmers have with profilers?
3. Is there a way to make using profilers less cognitive load on programmers?

Sometimes programmers fail to understand the cost of programs they write and need help writing more efficient code. Integrating profiling tools into the IDE could help them achieve their goal of being more productive programmers.

How does integrating a Python Profiling tool into an IDE help Python programmers of varied experience to detect unoptimized code more efficiently?

We find no prior art that looks at the user-experience. All papers on profiling and popular open source projects focus on technical ability of the data that the profiler can sample [2]. We believe that this is an area ripe for future research.

8 Conclusion

Although the results of our study were inconclusive, we believe that further work on the benefits of line-by-line in-IDE runtime feedback should prove its usefulness. We believe that although our study can't claim to contribute to the evidence in proving this, our study contributes to the asking of this vital question.

Our data suggest significant opportunities for improvement in the design of profiling tools. pytrail enables programmers to progressively evaluate and improve runtime issues. By further improving the performance tooling ecosystem, programmers can make software that saves energy, runs faster, and saves development time.

9 Acknowledgements

We would like to thank Michael Coblenz for creating this topics course, CSE 291[3]. Of course, this project would not exist without the contributors at Scalene and cProfile.

References

[1] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with Scalene."

[2] P. Kousha, B. Ramesh, K. Kandadi Suresh, *et al.*, "Designing a profiling and visualization tool for scalable and in-depth analysis of high-performance gpu clusters," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 93–102. DOI: 10.1109/HiPC.2019.00022.

3 Graduate Topics Course "Usability of Programming Languages" taught by Michael Coblenz, Fall 2023, https://cseweb.ucsd.edu/~mcoblenz/teaching/291A00_fall2023/