

Asking and Answering Questions During Memory Profiling

Alison Fernandez Blanco, Araceli Queriolo Córdova, Alexandre Bergel and Juan Pablo Sandoval Alcocer

Abstract—The software engineering community has produced numerous tools, techniques, and methodologies for practitioners to analyze and optimize memory usage during software execution. However, little is known about the actual needs of programmers when analyzing memory behavior and how they use tools to address those needs. We conducted an exploratory study (i) to understand what a programmer needs to know when analyzing memory behavior and (ii) how a programmer finds that information with current tools. From our observations, we provide a catalog of 34 questions programmers ask themselves when analyzing memory behavior. We also report a detailed analysis of how some tools are used to answer these questions and the difficulties participants face during the process. Finally, we present four recommendations to guide researchers and developers in designing, evaluating, and improving memory behavior analysis tools.

Index Terms—Program analysis, memory management, experimental design

1 INTRODUCTION

Developers often spend a substantial amount of time manually analyzing memory consumption to localize memory anomalies (e.g., memory leaks, memory bloats) that usually generate crashes on software applications [1], [2], [3]. For this reason, a number of memory profiling tools have been proposed to assist developers in this task, offering a wide range of information displayed through full-text reports or visualizations [4], [5], [6], [7]. Nevertheless, earlier studies have suggested that the information presented may be insufficient for programmers to identify and address memory issues [8], [9], [10]. Furthermore, other investigations also argue that how the information is displayed (whether as full-text or visualizations) impacts developers' comprehension during software analysis [11], [12], [13], [14].

A limited number of studies have presented empirical evidence regarding how memory profiling tools support developers in conducting memory analysis activities [15]. Understanding how programmers analyze memory behavior using tools and the challenges they encounter can provide valuable insights for the design and improvement of these tools. Consequently, there is still room for further research to enhance our understanding of the needs and behaviors of programmers when analyzing memory management.

In this paper, we undertook an exploratory study to provide a comprehensive and empirically-based set of questions that programmers ask during memory behavior analysis. In addition, we report on programmers' behavior when using two dedicated tools to answer these questions. We focused on understanding how programmers employ Python's memory

profiler tools because Python is considered one of the most popular programming languages¹ and it is primarily applied to Data Science and ML applications in academia, and several companies [16], [17]. The latter supports the intuition that Python programmers are more likely to analyze memory usage due to the large amount of data involved in their programming activities.

We selected two memory profilers, *Vismep* and *Trace-malloc*, for our study. These profilers provide diverse information through interactive visualizations and full-text reports, respectively. Memory profilers offer an extensive variety of features, and most profilers broadly differ on how information is provided and navigated. For example, some profilers [18], [19] may provide details about the garbage collector activity, while some others [20], [21] may focus on the context-call-tree or control flow. For this reason, selecting two different memory profilers instead of one hopefully enables us to cover different questions programmers ask. In addition, the selected memory profilers together provide a variety of features typically proposed by current memory profilers.

We observed twenty-two programmers analyzing software applications with which they were familiar, using the two memory profilers, and responding to open questions. We deliberately asked open questions to ensure participants looked for the information they considered valuable to understand memory behavior and detect optimization opportunities. Then, we centered on collecting and analyzing data about the questions asked by participants and how they employ the selected tools to answer those questions. For this, we followed a similar method of data extraction and analysis presented in other studies focused on identifying the information needs of developers [22], [23], [24]. One assumption we are making, based on our findings, is that memory profilers are tools relevant to people with moderate or extended backgrounds in addressing memory issues.

1. <https://insights.stackoverflow.com/survey/2019>

- A. Fernandez-Blanco, A. Queriolo-Córdova are with the ISCLab, Department of Computer Science (DCC), University of Chile, Beauchef 851, Santiago 8370456, Chile. A. Bergel is with RelationalAI (Switzerland). E-mails: afernand@dcc.uchile.cl, aqueirol@dcc.uchile.cl, alexandre.bergel@me.com
- J.P. Sandoval-Alcocer is with the Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile, Vicuña Mackenna 4860, Edificio San Agustín, 4to piso, Macul 7820436, Santiago, Chile. E-mail: juanpablo.sandoval@ing.puc.cl

This paper makes the following contributions:

Catalog of questions. We present an empirically-based catalog of 34 different questions asked by the participants. These questions are organized into five categories based on the information needed and the behavior of the programmer: understanding source code, understanding control flow, discovering the memory usage at a single point of time, comparing and contrasting memory consumption, and discovering memory events. To our knowledge, this is the most comprehensive list published to date that programmers may ask when analyzing memory behavior.

Tool usage analysis. We provide an observational analysis about how programmers employ *Vismep* and *Tracemalloc* to answer the raised questions. We discovered that participants numerous times combine multiple views (e.g., Call graph view and Source code view) from *Vismep* or use multiple features from *Tracemalloc* to obtain the required information. We also reported the questions that participants could not answer using these tools. Based on these results, we discuss the support missing from *Vismep* and *Tracemalloc*. Our analysis provides an opportunity to guide the design of tools to support programmers more effectively.

Outline. This article is structured as follows: we describe the essential background of the area, the context, and the current memory profilers for Python (Section 2). Section 3 outlines previous empirical studies about information needs in other domains. Section 4 details our exploratory study. Section 5 presents the 34 different questions and 775 question occurrences during the work sessions. Section 6 details how programmers employed the memory profilers to respond to the raised questions. Section 7 outlines unanswered question types and provides a number of recommendations. In Section 8, we discuss our findings and the open challenges. Finally, we close the paper with threats to validity in Section 9 and conclusion in Section 10.

2 BACKGROUND

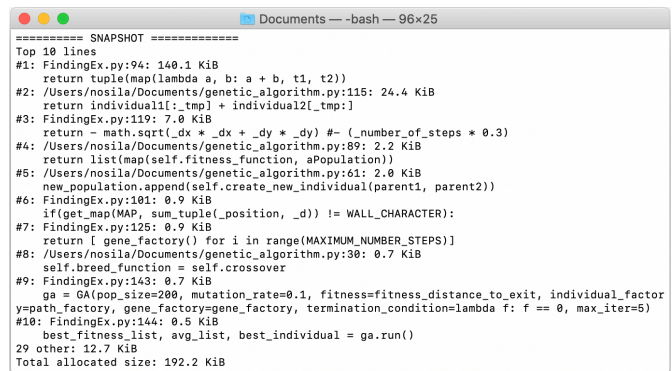
Memory space is a limited computational resource, and software applications need to allocate memory to store data values and data structures. Currently, software applications handle a vast amount of data (e.g., data science, artificial intelligence) that increases over time. Thus, understanding memory behavior is vital to locating and repairing memory issues that could cause performance degradation and program crash [1], [3]. For example, when programmers locate an abnormal memory growth during software execution, they usually report “consistently observing memory growth” or “can lead to a leak” to indicate potential issues [25]. An example of a memory anomaly is memory bloat which happens when a program requires excessive memory to operate correctly, exposing inefficient memory usage of a program [26], [27]. Consequently, programmers may locate and investigate the code responsible for suspicious memory allocations. Another memory anomaly is a memory leak often occurring when an unused memory resource is not released [25], [28], [29]. Therefore, programmers could explore when and why the memory is not released.

Manually performing these activities is complex as programmers must (i) analyze multiple technical memory

aspects simultaneously, e.g., memory allocations, garbage collector information and (ii) carefully examine the corresponding code and memory behavior [6], [30].

2.1 Memory Profiling

Mainstream modern software development environments provide memory profiler tools to support practitioners monitoring memory usage during software execution. These profilers assist developers in evaluating how well programs perform based on a set of aspects, including memory usage, garbage collections, and frequency of function calls [6], [7], [26], [31]. For this, they first help programmers extract valuable information from program execution. Then, the profiler presents the extracted information through full-text reports, textual tables, and visualizations.



```
Documents -- -bash -- 96x25
===== SNAPSHOT =====
Top 10 lines
#1: FindingEx.py:94: 140.1 KiB
return tuple(map(lambda a, b: a + b, t1, t2))
#2: /Users/nosila/Documents/genetic_algorithm.py:115: 24.4 KiB
return individual1[:_tmp] + individual2[:_tmp]
#3: FindingEx.py:119: 7.0 KiB
return - math.sqrt(_dx * _dx + _dy * _dy) #-( _number_of_steps * 0.3)
#4: /Users/nosila/Documents/genetic_algorithm.py:89: 2.2 KiB
return list(map(self.fitness_function, aPopulation))
#5: /Users/nosila/Documents/genetic_algorithm.py:61: 2.0 KiB
new_population.append(self.create_new_individual(parent1, parent2))
#6: FindingEx.py:101: 0.9 KiB
if(get_map(MAP, sum_tuple(_position, _d)) != WALL_CHARACTER):
#7: FindingEx.py:125: 0.9 KiB
return [ gene_factory() for i in range(MAXIMUM_NUMBER_STEPS)]
#8: /Users/nosila/Documents/genetic_algorithm.py:30: 0.7 KiB
self.breed_function = self.crossover
#9: FindingEx.py:143: 0.7 KiB
ga = GA(pop_size=200, mutation_rate=0.1, fitness=fitness_distance_to_exit, individual_factor
y=path_factory, gene_factory=gene_factory, termination_condition=lambda f: f == 0, max_iter=5)
#10: FindingEx.py:144: 0.5 KiB
best_fitness_list, avg_list, best_individual = ga.run()
29 other: 12.7 KiB
Total allocated size: 192.2 KiB
```

Fig. 1. Enlisting the top ten sites that allocate most memory using Tracemalloc.

Reported information. The data extracted by memory profilers varies [15]. For example, memory profilers often collect the memory allocations made during software execution along with the software components (e.g., function or method) responsible for these allocations to support users with the detection of allocation hotspots and memory growth [6], [7], [31]. Some profilers add metrics about the release of memory [4], [5], [32], [33], and the references between the objects allocated to assist programmers in discovering memory leaks [34], [35], [36]. Furthermore, a number of profilers connect the dynamic aspects with the source code to help developers locate the cause of an issue [6], [26].

Report presentation. Memory profilers show the collected information in diverse ways. Several profilers display lists or tables through a full-text report or non-interactive visualizations. An example is Tracemalloc, which enlists the primary sources (code lines and files) responsible for allocating the most memory (see Figure 1). Another typical example is Yourkit [37], and JProfiler [38], which report data structures that consume the most memory, the number of instances of the data structure, and the amount of allocated memory for each type using a textual table. On the other hand, various studies introduce profilers that report information using interactive visualizations to facilitate the practitioner’s comprehension [39], [40]. For instance, some interactive visualizations [6], [7], [31], [36], [41], [42] support developers with the memory consumption analysis for C/C++ and Java programs using different visualization techniques (e.g., node-link diagrams, hierarchical stacking) [15], [43].

Evaluation of tools. Most proposed studies evaluate memory profiler tools through usage scenarios [5], [7], [31], [44]. Few studies present empirical evidence about how the tool performs with programmers and software applications [15]. These studies primarily aim to assess tool usefulness for practitioners, highlighting valuable features for programmers knowledgeable in memory management [6], [45]. For instance, Weninger *et al.* [26] conducted a qualitative study to analyze students' behavior using AntTracks to address memory anomalies and providing general recommendations for researchers and developers of interactive memory analysis tools. Fernandez *et al.* [21] performed an exploratory study evaluating Vismep's support in memory usage analysis, reporting five information needs and how programmers utilized Vismep to fulfill them.

In contrast to the previous studies, our work provides a catalog of questions developers ask themselves when understanding the memory behavior of Python applications. Furthermore, we detail how programmers employ two memory analysis tools to answer these questions and discuss the difficulties encountered while using these tools.

2.2 Memory Consumption Analysis in Python

Several approaches center on memory usage analysis in Python. Table 1 illustrates the tools/libraries along with the (i) activities they claim to support, (ii) information collected, and (iii) report presentation used. We extracted this information and other data (*e.g.*, installation requirements, links) from their respective documentation².

These approaches extract diverse information and report it using textual reports, non-interactive visualizations, and interactive visualizations. The libraries in Table 1 are highly expressive, flexible, and can generate tuned reports (primarily textual), but users must modify their code using the correct function calls. For instance, *memory_profiler* [49] provides a decorator (`@profile`) to mark the functions to be profiled and report the memory used by each code line from the selected function. On the other hand, standalone tools [19], [20], [21] have a graphical user interface or a command-line interface that allows users to extract and report specific information without manually changing the code.

Both libraries and tools commonly claim to help programmers in the following activities:

Analyzing Allocations and Allocation Sites. Most approaches report memory allocations during program execution, displaying allocations made by specific parts of the code, such as variables, lines of code, or functions. For instance, tools like *Muppy*, *Guppy*, and *vprof* provide information about memory allocations and the memory occupied by these allocations during program execution. Other approaches [18], [20], [49], [21], [50], [51] focus on displaying the executed functions and the associated memory consumption per function or line-by-line.

Analyzing memory usage over time. Some approaches [19], [49], [50], [51] display the memory usage over time through line charts or sparklines. Although other memory profilers [18], [46], [47], [21] do not collect any time metric, they help

users note changes in memory usage between points in time. For example, *Tracemalloc* shows how memory usage changes (increased, decreased) before and after executing a function. In addition, *Fil*, *Vismep*, *memray* highlight the code (lines of code, functions) responsible for allocating most memory³ using visual hints (*e.g.*, color, size). To exemplify, *Vismep* provides a visualization representing each executed function as a rectangular box, where the width indicates the memory allocated by an invoked function during program execution. Therefore, programmers could visually detect the function responsible for allocating the most memory by locating the widest box in the view.

Analyzing leaking objects. Few libraries [46], [47], [48] show the memory allocations made at a given point in time (after a function call) along with the objects that reference these allocations. For instance, *objgraph* displays a graph where each node is an object allocated in memory, and the edges represent the references between objects. These libraries allow programmers to detect memory leaks by locating unused memory resources.

Impact. Although several libraries and tools are available to help programmers with memory usage analysis, some are limited due to their installation requirements, documentation available, maintenance, and problems with functionality. Additionally, when practitioners adopt libraries, they need to make the right function calls to obtain the required data and filter only relevant values (*e.g.*, excluding data structure directly allocated by the runtime and not the profiler application). Furthermore, both standalone tools and libraries may have limitations in the information reported and their interaction mechanisms. Consequently, practitioners could face challenges when adopting some of these options.

3 RELATED WORK

Several studies centered on extracting information about developer needs. As a result, these studies usually present emerging questions raised by developers in particular scenarios.

Sillito *et al.* detect 44 types of questions asked by programmers during software evolution tasks [23], [24]. These questions are categorized based on the kind and scope of the required information. The study also exposes that developers need better tool support to answer some specific questions. In a similar field, Kubelka *et al.* [22], [52] analyze the impact of live programming when developers perform software evolution tasks and identified eight additional questions compared to the set of questions provided by Sillito and colleagues [23], [24]. Additionally, Kubelka *et al.* notice that Live Programming impacts the questions asked by developers and the use of tools.

LaToza *et al.* survey professional developers to identify questions about code that they consider hard to answer [27]. They gathered 94 questions grouped into 21 categories. They reported that the most frequent questions deal with the intent and rationale of code or describe specific situations, such as security or error logging.

3. We use the term *memory allocated by a code entity* to refer to the memory allocated during the execution of the respective entities, such as lines of code or functions.

2. <https://www.dropbox.com/s/49mdqg5n11bhvdd/PythonMemoryProfilers.csv?dl=0>

TABLE 1

Libraries and tools along with (i) the activities that claim to support (A1 = Analyzing allocations and allocation sites; A2 = Analyzing memory usage over time; A3 = Analyzing leaking objects), (ii) the information reported (M.A. = Memory allocations; M.R = Memory releases; R.F. = Relationships between functions; O.R. = Object references; T. = Time; TH. = Threads; L.C. = Lines of code; C. = Class; S.C. = Structural component) and (iii) the report presentation used. Accessed at 02/09/2022.

Library/Tool	Activities			Information reported									Report presentation	
				Program execution					Source code					
	A1	A2	A3	M.A.	M.R.	R.F.	O.R.	T.	TH.	L.C.	C.	S.C.	Textual	Visualization
Guppy [46]	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
Muppy [47]	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
Objgraph [48]	✓	✗	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓
Memory_profiler [49]	✓	✓	✗	✓	✓	✗	✗	✓	✗	✓	✗	✗	✓	✓
Tracemalloc [18]	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✗	✓	✓	✗
Fil [20]	✓	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓	✗	✓
vprof [19]	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓
Vismep [21]	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✗	✓	✗	✓
Scalene [50]	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓
memray [51]	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

Fritz *et al.* identify 78 questions raised by developers during a software development project. They also report a lack of tool support to answer these questions [53] since these questions involves connecting information from different sources (*e.g.*, source code, changesets, teams).

De Alwis *et al.* collect 36 questions from literature, blogs, and their experience in software development [54]. They claim that developers present difficulties when answering some questions because connecting multiple results from different tools is necessary to extract the required information.

Ko *et al.* detect 21 types of questions when analyzing software development teams [55]. The work highlights that the most frequent questions are related to mistakes in the code and co-workers' activities.

LaToza *et al.* focus on reachability questions and enlisted 12 questions with their difficulty and frequency [56]. The results show that reachability questions are challenging to answer and are associated with the most prolonged activities. Due to this, tools with support to answer these questions are relevant.

In contrast to the studies previously mentioned, to our knowledge, our work is the first observational study centered on developer information needs during memory consumption analysis.

Impact. We consider that documenting solid knowledge about programmers' needs while monitoring memory behavior helps (i) improve the design and effectiveness of the current tools and new ones, (ii) recognize if a tool fits the programmers' needs and which needs may not currently be covered and (iii) facilitate the organization of current approaches to help practitioners find a suitable tool.

4 METHODOLOGY

Our study investigates the questions Python programmers raised during memory usage analysis and how programmers answer these questions using memory profiler tools. Consequently, we designed an exploratory study as Figure 2 illustrates. The following subsections describe the main steps of the study.

4.1 Research Questions

We aim to answer the following research questions (RQ):

- **RQ1:** *What questions do Python programmers ask when analyzing memory behavior using memory analysis tools?*
- **RQ2:** *How do Python programmers answer these questions using memory analysis tools?*
- **RQ3:** *Which type of questions did Python programmers not answer, and what barriers did they face?*

Our research questions focus on three dimensions: information needs (**RQ1**), tool usage (**RQ2**), and the challenges faced (**RQ3**). Our three research questions were inspired by previous studies that reported information needs in other context domains (see Section 3) and studies that propose memory analysis tools (see Section 2.1).

RQ1 centers on identifying programmer's questions during memory behavior analysis to discover valuable information needs that researchers and tool builders should consider when designing adequate tools. Compared to previous literature (Section 3), our study focuses on information needs during memory behavior analysis.

RQ2 focuses on understanding how programmers use current approaches to satisfy their information needs. Therefore, we provide empirical evidence about the practical support of approaches (*e.g.*, useful features, helpful information) and the actions made during the process. **RQ3** highlights the challenges users face when using memory analysis tools to obtain the required information. The latter helps identify inadequate or missing features of the approaches and provides opportunities to improve the tools.

4.2 Memory Profiler Tools Under Study

We selected Tracemalloc and Vismep to understand the impact of memory profiler tools on supporting programmers with memory behavior analysis.

Tracemalloc. Tracemalloc is one of the most flexible libraries and provides multiple functions to display information related to memory usage through full-text reports. Programmers must modify the application's code under study when using Tracemalloc [18]. Tracemalloc is part of the standard distribution of Python. It provides several features:

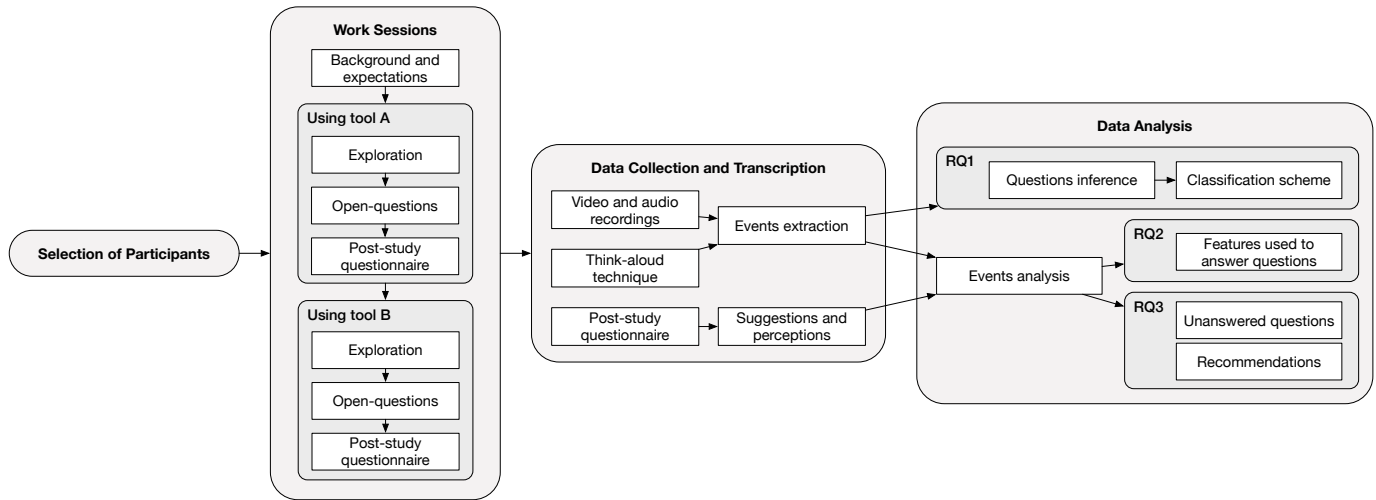


Fig. 2. Overview of the workflow of the exploratory study.

- *Display TOP*. This feature assists programmers in enlisting the top sites (code line, file) responsible for allocating the most memory, as shown in Figure 1.
- *Compute differences*. Tracemalloc supports programmers in exploring memory usage over time by indicating the differences (increase, decrease) in memory consumption before and after executing the potential leaking function.
- *Get traceback*. This feature helps programmers traceback where an object was allocated, specifically, in which case the memory allocation is made.
- *Get traced memory*. Tracemalloc presents functions to display the total memory occupied and the peak during the computation of certain parts of code.

Vismep. Vismep is an interactive visualization for supporting programmers in analyzing memory usage over Python applications [21]. To employ Vismep, a developer must execute scripts with specific parameters. Vismep collects the invoked functions with their respective memory footprint during the program execution. Vismep also gathers the calling relationships between invoked functions. To display this information, Vismep provides multiple views as shown in Figure 3:

- *Call graph view*. The main view summarizes the functions with the calling relationships and the memory footprint using an interactive call graph. This view assists users in locating relevant code, detecting allocation sites responsible for memory usage increment, and identifying the circumstances in which memory is allocated.
- *Source code view*. This view displays the source code of a function and highlights the code lines responsible for memory usage increment. It supports practitioners in detecting memory growth at a more fine-grained level (code line) and understanding the memory events involved with a particular code.
- *Scatter plot view*. Vismep provides a secondary view that shows the relationships between memory behavior and the execution times of functions. This view assists programmers in exploring memory growth and learning how the memory is used (allocated/released) during the program execution.

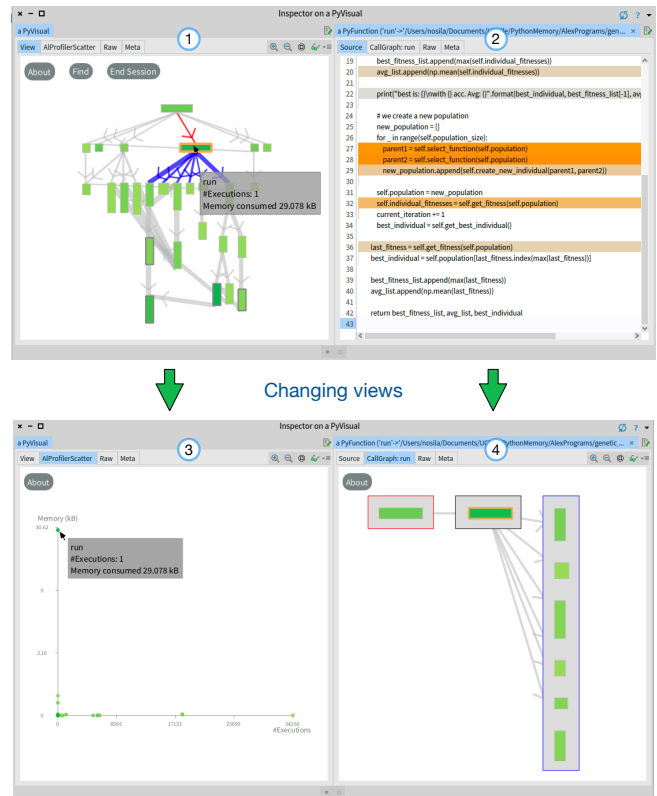


Fig. 3. Visualizing an example with Vismep (1) Call graph view – the main view that summarizes the functions with the calling relationships and the memory footprint, (2) Source code view – a view that displays the source code and highlights memory usage increment, (3) Scatter plot view – a secondary view that shows the relationships between memory behavior and the execution times of functions, (4) Sub call graph view – for navigating through the calling relationships of a function.

- *Sub call graph view*. Vismep facilitates users in navigating a selected function's direct calling relationships. As a result, the user can select an allocation site (function) and navigate through an execution path.

Vismep also offers several interaction mechanisms, such

as canvas movement (*e.g.*, panning, zoom in and out), search option (find a function based on its name), and options to obtain detailed data about a particular function (*e.g.*, popup window with information-on-demand).

Selecting memory profilers. We selected Tracemalloc and Vismep for several reasons:

- *Reported information.* Tracemalloc and Vismep provide information to perform various tasks described in Section 2.2 (*e.g.*, analyzing memory growth). Furthermore, they connect dynamic information (*e.g.*, memory allocations) with the source code. Therefore, we consider that they help programmers trace memory events and better understand program behavior.
- *Report presentation.* Tracemalloc provides only full-text reports, which has become standard among other options for profiling. On the other hand, Vismep reports the information only using interactive visualizations that could facilitate data comprehension [39], [40]. Consequently, we considered investigating how programmers employ both approaches.
- *Availability and maintenance.* Tracemalloc and Vismep are available, maintained, and provide material (structured documentation, examples) for practitioners to learn how to use them. Tracemalloc is a native module⁴ used internally in Python to improve other functionalities (*e.g.*, `ResourceWarning` reports). As a result, it works independently of the operative system and has no external dependencies. Furthermore, a previous research introduced Vismep and presented how Vismep support programmers with memory usage assessment [21].

Vismep and Tracemalloc offer features commonly found in various Python memory profiler tools. Therefore, we expect that selecting these two different memory profilers will increase the diversity and range of the questions asked by practitioners. However, as indicated in Table 1, the selected tools lack certain runtime information, such as object references and multithread analysis. The analysis of object references has proven valuable for detecting memory leaks and optimization opportunities. Similarly, considering threads in memory analysis is essential for specific application types [50]. Nevertheless, only a few Python approaches provide this information compared to other programming languages [15], [26]. In our future work, we plan to explore additional tools that further enhance the scope of our study.

4.3 Participants & Applications

To recruit participants, we sent invitations to students and bachelors from our university and members of Python communities. We make it clear in the invitation that the study investigates how programmers analyze the memory behavior of familiar code using memory profiler tools. We also clarified that programmers who wished to participate in the study must choose a Python program they have written themselves and intend to analyze its memory behavior.

Participant Selection. We carefully selected twenty-two programmers who volunteered to participate and have a Python application to analyze in the study. This participant group comprises individuals from diverse fields of study. We reason

that numerous Python programmers (*e.g.*, data scientists, journalists) may not possess or actively pursue a formal Computer Science degree. Specifically, eleven participants have or are pursuing a degree in Computer Science, while the rest come from diverse fields such as Geology, Mathematics, and Electrical Engineering. Five participants were from the industry, four were in research centers, four pursued a master's degree, and the rest were in their bachelor's studies. Six participants were women. Table 2 details the demographic information of each participant.

Experience in Python programming. Participants manifested various experience levels in software development, but all were familiar with Python programming. Their average experience in programming with Python was 4.70 years (std. dev. 2.24). Also, participants self-assessed their experience using a Likert scale of five steps, *i.e.*, 1 (novice) to 5 (expert). As a result, the average experience in Python programming was 3.29 (std. dev. 0.73).

Experience in memory usage analysis. We intentionally incorporated participants with diverse experiences in addressing memory issues (*e.g.*, memory bloats, leaks, churn) and performing memory analysis activities. Based on previous studies [15], [26], we categorized participants' experience into two activities:

- *A1: Analyze memory behavior in a single point.* All participants mentioned that they frequently focused on memory allocation details in various parts of the code, such as data structures, functions, or specific lines, at specific moments during execution. For instance, P3 mentioned, "I usually check where I create objects in the code to see how they affect memory behavior and if they're really needed in that specific moment".
- *A2: Analyze memory behavior over time.* Thirteen participants mentioned that they often check how memory is handled (used and released) over time to detect memory leaks and excessive memory usage.

While all the participants have experience analyzing memory behavior, fourteen have experience fixing memory issues. Although the remaining eight have not formally resolved memory issues, they perform memory analysis activities while developing their applications. Various memory analysis experiences enhance our capacity to gather multifaceted insights into memory analysis questions. It also helps provide a deeper understanding of how distinct participants use the tools under study. Table 2 provides an overview of participants' involvement in memory issue resolution and related activities. Furthermore, we observed that participants usually modify their code to perform these activities and identify abnormal memory behavior using functions from specialized libraries.

Studied applications. As mentioned before, we explicitly asked participants to choose a Python application to analyze during the study. Since monitoring memory behavior is not a trivial activity, we suggest that participants select an application they are familiar with (own code, project) and find interesting to analyze in terms of memory consumption. Consequently, participants selected different programs (*e.g.*, data analysis, IA, ML) with which they were familiar. Additionally, they mentioned that their selection was based on either (i) they considered memory usage a potential threat

4. <https://python.readthedocs.io/en/stable/whatsnew/3.6.html>

TABLE 2

Information of participants (Python programming experience (years); Self-assessment expertise (Likert-scale: 1 (novice) to 5 (expert)); Experience in memory behavior analysis; Experience in fixing memory issues; Activities when analyzing memory behavior). Participants from groups G1 and G2 present gray and white backgrounds, respectively.

ID	Study Field	Python Programming		Experience in Memory Behavior Analysis and Issues		
		Experience (Years)	Self-assessment Expertise	Memory Usage Analysis	Fixing Memory Issues	Activities Performed
P1	Geology	9	2.5	✓	✓	A1, A2
P2	Electrical Engineering	4	3.5	✓	✓	A1, A2
P3	Electrical Engineering	6	4	✓	✓	A1, A2
P4	Physical Engineering	5	3	✓	✓	A1, A2
P5	Electrical Engineering	8	5	✓	✗	A1
P6	Aerospace Engineering	2.5	3	✓	✗	A1
P7	Computer Science	1.5	3	✓	✓	A1, A2
P8	Computer Science	6	4	✓	✓	A1
P9	Computer Science	4	3	✓	✓	A1, A2
P10	Computer Science	5	4	✓	✗	A1
P11	Computer Science	3	3	✓	✗	A1
P12	Metallurgical Engineering	1	3	✓	✓	A1
P13	Electrical Engineering	8	3.5	✓	✓	A1, A2
P14	Pedagogy in Math and Computing	3	2	✓	✓	A1
P15	Mathematical Engineering	3	3.5	✓	✗	A1, A2
P16	Pedagogy in Math and Computing	1	2	✓	✗	A1
P17	Computer Science	7	4	✓	✓	A1, A2
P18	Computer Science	5.5	4	✓	✓	A1, A2
P19	Computer Science	5	2.5	✓	✓	A1, A2
P20	Computer Science	6	4	✓	✓	A1, A2
P21	Computer Science	4	3	✓	✗	A1, A2
P22	Computer Science	6	3	✓	✗	A1

to their application or (ii) they wanted to verify assumptions about memory usage and find ways to reduce memory consumption.

4.4 Procedure

We conducted a work session for each participant with her/his respective application. Each work session started with a moderator explaining the study's goals described in the invitation to programmers who agreed to participate in the study. The moderator also explained how the think-aloud protocol [57] works and asked the participant to use it during the session. Then, general questions are asked to the participant to collect demographic data such as age, gender, experience in Python programming, memory usage analysis, and addressing memory issues. After these questions, the participant describes her/his application and gives an opinion about the application's memory usage. The participant also explains the expectations or assumptions about elements (*e.g.*, functions, instances) that may produce a memory anomaly (*e.g.*, memory bloat, leak) during the program execution.

Furthermore, the session proceeded with two phases, each with a different memory profiler tool. Both phases are structured as follows:

- 1) *Exploration*. The participant read the documentation about the memory profiler tool and had an exploration phase to familiarize herself/himself with the tool. The participant also had the opportunity of asking the moderator questions about the tool or its documentation.
- 2) *Open-questions*. The participant employed the respective memory profiler tool for analyzing the memory usage of her/his application and answered the open questions

in Table 3. As stated before, we deliberately asked open questions to ensure that participants had a goal they cared about and looked for the information they considered valuable to understand memory usage and detect optimization opportunities.

- 3) *Post-study questionnaire*. The participant answered verbally and informally open questions regarding their observations, recommendations, and perceptions of the memory profiler tool.

For our study, we randomly divided the participants into two groups, G1 and G2, each containing eleven participants. These groups are balanced with participants from the computer science field and other fields (*e.g.*, electrical, mathematical engineering). Table 2 presents the participants from G1 and G2 with gray and white backgrounds, respectively. In G1, participants initially worked with Vismep and then with Tracemalloc, while in G2, the order of tools was reversed. In both phases, participants analyzed the same application they selected. As mentioned above, we examine the benefits and drawbacks of tools' features when analyzing memory behavior since they report information using different approaches. Note that our goal has not been to compare Vismep and Tracemalloc.

We recorded a video of the screen and the laptop's audio used during the work sessions. These recordings were used to collect data and analyze participants' reasoning process.

4.5 Data Collection and Transcription

To answer our research questions, we aim to identify questions asked by the practitioners during memory consumption analysis (RQ1), how they employ memory tools to answer these questions (RQ2), and detect the barriers faced in the

TABLE 3
Questions made to the participant during the work session.

Open Question	Rationale
Q1: <i>Can you characterize the memory consumption of your application?</i>	The participant identifies the information relevant for memory usage analysis, such as memory growth, the allocation hotspots or the allocations made during program execution.
Q2: <i>What have you learned from your application? Anything surprising?</i>	The participant compares the profiler's report with their assumptions, identifies additional or unknown valuable information, and highlights potential issues within the application.
Q3: <i>Do you find an opportunity to decrease memory consumption? If yes, can you improve it and run the profiler again?</i>	The participant describes and discusses the opportunities to reduce the application's memory usage if any. The participant also modifies the code that could be the cause of a memory anomaly. Then, the participant verifies the impact of the changes in the application's memory usage using the tool.

process (RQ3). To achieve our goal, we extracted information from work sessions when participants employed the memory profiler tools to answer the open questions in Table 3.

Firstly, we reviewed and checked session video and audio recordings to generate spreadsheets that summarize each work session. Each spreadsheet contains: (i) the memory profiler (Vismep or Tracemalloc) used, (ii) the open question that the participant responded, (iii) the respective period of time in the video record, (iv) the verbalized thoughts of the participant, (v) the actions made by the participant, and (vi) the memory profiler tool features used.

To minimize biases in the data collection process, one author generated the spreadsheets, and another author checked if the data from the spreadsheets was consistent with the audio and video records.

4.6 Data Analysis

This section describes the methods employed to analyze the collected data.

Questions inference. To identify the questions asked by the participants during our study, we performed an analysis method similar to the one proposed by Kubelka and colleagues [22]. This method consists of two steps: (i) identifying the concrete questions by analyzing the video and (ii) generalizing, encoding, and unifying the concrete questions.

Firstly, we detected concrete questions using the spreadsheets generated for each work session. Some questions were inferred based on the actions and verbalized thoughts of the participants. For example, we inferred *“What part (function, line of code, instance) of main function consumes the most memory?”* as the concrete question for the actions mentioned in the following example:

Tool: Vismep
Open-question: Q1
Time: 00:01:40 - 00:01:53
Verbalized thoughts: For the `main` function, the lines that consume much memory are reading the file, detecting the fire, and showing the points where the fire is detected.
Participant actions: The participant observes the source code of `main` function and jumps to the highlighted lines of the view while pointing out and describing the code lines.
Features used: Source code view
Inferred question: What part (function, line of code, instance) of `main` consumes the most memory?

We also gathered concrete questions that the participants directly mentioned. To illustrate, P6 passed the cursor over the `CALC_PARAMETERS` function and examined the connected nodes with blue edges (outgoing functions) while he asked, *“Which functions does `CALC_PARAMETERS` call?”*.

Then, we defined generic questions based on concrete questions to abstract the details of a given task. For this, we generalized the questions by identifying similar concrete questions. For instance, we inferred the generic question *“Which functions does this function call?”* from the concrete question *“Which functions does `CALC_PARAMETERS` call?”* to reference any function on the execution of a program. We also mapped some questions with the questions mentioned by Sillito *et al.* [23], [24] and Kubelka *et al.* [22] due to the presence of questions related to understanding the control flow and the rationale behind the source code. For example, we transcribed *“Where is this method called or type referenced?”* proposed by Sillito *et al.*, instead of our generic question *“Which functions call this function?”*. We noticed that the participants asked these questions to enrich their comprehension of the software application and make decisions about memory anomalies or opportunities for improvement in memory usage.

To minimize biases in the data analysis, one author inferred the concrete and generic questions, and another checked if the inferred questions (concrete and generic) were consistent with the information in spreadsheets. In addition, all the authors held two meetings to discuss any discrepancy in the inferred questions' consistency.

Classification scheme creation. We opted to organize the discussion of the inferred questions around a classification scheme. To create this classification scheme, the first author conducted a thematic analysis [58] by following these steps:

- **Familiarization.** The inferred questions were read and reread to obtain an overview of the data.
- **Generating codes.** To reflect relevant features of each question, the author in charge assigned codes. For instance, the author assigned the code *“Understanding implementation”* for the question *“Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?”*. Furthermore, the author conducted continuous reviews to refine and check the consistency of the assigned codes.
- **Constructing initial themes.** The author created coherent groups to identify broader patterns based on the assigned codes with their associated inferred questions. If a code does not belong to a specific theme, it is assigned

to a miscellaneous group and analyzed later.

- *Reviewing themes.* The author checked the initial themes against the inferred questions to refine and create the final themes.
- *Defining and naming themes.* Final themes were defined with a descriptive name and a detailed description.

Additionally, two authors reviewed the consistency of codes and themes against the associated data. Two meetings were held involving three authors to discuss the disagreements or potential issues of the generated codes and themes. Consequently, we minimized potential inconsistencies in the coding process.

Furthermore, an author and a professional software engineer independently categorized the inferred questions using the generated classification scheme to validate the reliability. For this, each one filled out a spreadsheet to categorize the inferred questions based on the detailed description of the created categories. Subsequently, we calculated the Cohen kappa [59] as a reliability metric to examine the agreement between reviewers. As a result, we detected that reviewers present an “almost perfect agreement” ($\kappa > 0.81$). The latter suggests that the classification scheme presents accurate representations (themes) for the inferred questions.

Events and questions analysis. To answer RQ2 and RQ3, we first extracted the actions made by the participants and the features used to answer the inferred questions. Then we gathered patterns by concentrating on frequent actions performed to answer specific questions using certain features of memory profiler tools. We also located questions that participants declared they could not answer due to difficulties when using the memory profiler tools.

We considered whether or not the question asked was answered based on the verbalized thoughts of the participants. For instance, participants suggest that questions were not answered when they mentioned phrases like “I can't find out with the information I have” or “I think this change would reduce memory but I can't do it right now because it will take a lot of time and it's complex”.

5 RQ1: WHAT QUESTIONS DO PYTHON PROGRAMMERS ASK DURING MEMORY BEHAVIOR ANALYSIS?

We identified 34 different questions and 775 question occurrences from 22 work sessions, with a total duration of 24 hours (exploration, open-questions, and post-study questionnaire). Also, we generated a classification scheme with five categories to organize these inferred questions.

Table 4 illustrates the distribution of questions per category and the occurrences raised by the participants during the work sessions. The first column provides the questions categorized according to our classification scheme. The following twenty-two columns summarize the questions occurrences during each work session with the format: #Total questions occurrences (#Occurrences using Vismep / #Occurrences using Tracemalloc) if any occurrence, otherwise the cell is empty. We also use the cell background to show the relative frequency of the questions, in which darker green backgrounds indicate the most frequent questions in the sessions. Additionally, the last column follows the format previously described and represents the total number of question occurrences per question or category.

The following sections describe each category and the questions related to them.

5.1 Understanding Source Code

This category involves questions centered on understanding aspects of the source code, such as its static structure, rationale, and implementation. We noticed that in this category, six questions and 158 question occurrences (20%) were raised during the work sessions. The following questions are most frequent in this category: “4. What does the declaration or definition of this look like?”, “1. Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?”, and “2. Which entities (functions, lines of code, class) belong to this file or module?”

Note that the questions in this category may not initially align with memory analysis. However, we must consider that memory analysis, like any other program comprehension activity, involves navigating and understanding the source code. Upon reviewing the literature, we found that five of the six questions in this category resemble questions asked by programmers in other contexts, such as programming change tasks [22], [23], [24], [52]. Specifically, during the sessions, participants posed these questions when (i) searching for a particular piece of code to analyze its memory usage or (ii) identifying a memory issue or opportunities for memory optimization.

Finding a particular code. When participants characterize the memory behavior of their applications, they commonly focus on finding entities (functions, instances) relevant to the program functionality based on their knowledge to analyze its memory usage. For example, P3 wanted to analyze the memory usage of some auxiliary functions that he considered relevant: “I want to find my auxiliary functions that performs multiple calculations and I want to know how much memory they consume”. Participants usually found relevant entities based on the provided behavior, the module they belong to, and the entity's name. As a result, the next questions were asked: “1. Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?”, “2. Which entities (functions, lines of code, class) belong to this file or module?”, and “3. Is there an entity named something like this in that unit (project, package, or class)?”

Locating anomalies and improvements. When participants tried to determine if an entity (function, line of code) was involved with an anomaly (e.g., memory bloat, leak), they often explored the entity's source code to decide if the memory allocated was necessary or not for the correct functionality of the program. Furthermore, when participants locate an anomaly or an opportunity for reducing memory usage, they center on the parts of code responsible for the anomaly to investigate how it is defined or structured and what the code is supposed to do. Consequently, participants asked the following questions: “4. What does the declaration or definition of this look like?”, “5. What are the parts of this entity (function, instance, type)?”, and “6. What is the behavior that these entities (functions, lines of code, instances) provide together?”

5.2 Understanding Control Flow

Questions related to understanding the control flow (e.g., exploring the relationships between functions, identifying

TABLE 4

Questions per category and the occurrences raised during the work sessions. Each column corresponding to a participant presents the format: T (A/B), where A denotes the number of occurrences using Vismep, B indicates the number of occurrences using Tracemalloc, and T denotes the total number of occurrences. Note that if T is zero, the cell is empty.

Questions Type per Category	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	Total
Understanding Source Code																							
1. Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?	2 (2/0)	1 (1/0)	2 (2/0)	2 (2/0)	1 (1/0)	1 (1/0)	2 (1/1)	1 (1/0)	1 (1/0)	2 (2/0)	1 (1/0)	2 (2/0)	2 (2/0)	1 (1/0)	2 (2/0)	2 (2/0)	1 (1/0)	2 (2/0)			3 (3/0)	2 (2/0)	33 (32/1)
2. Which entities (functions, lines of code, class) belong to this file or module?	4 (1/3)	1 (0/1)		3 (2/1)	2 (0/2)	2 (1/1)	1 (1/0)	2 (1/1)	3 (2/1)			3 (2/1)	2 (2/0)	2 (2/0)			1 (1/0)	2 (1/0)	1 (0/1)		4 (3/1)		33 (19/14)
3. Is there an entity named something like this in that unit (project, package, or class)?			1 (1/0)	2 (2/0)	1 (1/0)	1 (1/0)							6 (6/0)	2 (2/0)			2 (2/0)	1 (1/0)				1 (1/0)	17 (17/0)
4. What does the declaration or definition of this look like?	4 (0/4)	7 (1/6)	3 (2/1)	2 (2/0)	2 (0/2)	4 (2/2)	2 (2/0)	1 (0/1)	4 (3/1)			5 (5/0)	3 (3/0)		1 (1/0)			1 (1/0)	5 (1/4)	1 (1/0)	1 (1/0)		46 (26/20)
5. What are the parts of this entity (function, instance, type)?							2 (2/0)																2 (2/0)
6. What is the behavior that these entities (functions, lines of code, instances) provide together?	2 (1/1)				2 (1/1)	1 (1/0)	2 (2/0)	2 (2/0)	3 (1/2)	1 (0/1)		1 (1/0)	1 (0/1)	3 (2/1)			1 (0/1)		7 (2/5)		1 (1/0)		27 (14/13)
Total in the category																							
	12 (4/8)	9 (2/7)	6 (5/1)	9 (8/1)	8 (3/5)	9 (6/3)	7 (6/1)	8 (7/1)	11 (7/4)	3 (1/0)	1 (1/0)	11 (10/1)	14 (13/1)	3 (3/0)	8 (7/1)	2 (2/0)	5 (4/1)	6 (5/1)	13 (3/10)	1 (1/0)	9 (8/1)	3 (3/0)	158 (110/48)
Understanding Control Flow																							
7. Which entities (functions, lines of code) are the most executed?	1 (1/0)	1 (1/0)	3 (3/0)	1 (1/0)	2 (2/0)	2 (2/0)		1 (1/0)		1 (1/0)	1 (1/0)		2 (2/0)										15 (15/0)
8. Where is this method called or referenced?	1 (0/1)	1 (1/0)	1 (1/0)		2 (2/0)		1 (1/0)		1 (0/1)				1 (1/0)		1 (1/0)			1 (1/0)					10 (8/2)
9. When during the execution is this method called?	2 (0/2)			1 (0/1)	1 (1/0)	1 (0/1)																	5 (1/4)
10. Which functions are called by this function?			2 (2/0)		2 (2/0)	4 (4/0)	1 (1/0)		2 (2/0)			1 (1/0)	1 (1/0)	1 (1/0)	1 (1/0)				1 (1/0)	4 (4/0)		3 (3/0)	1 (1/0)
11. How many times is this entity (function or line of code) executed?	2 (0/2)	2 (0/2)	2 (2/0)		8 (8/0)	1 (1/0)	2 (2/0)	2 (2/0)	2 (2/0)		2 (2/0)					1 (1/0)							26 (22/4)
12. How many recursive calls happen during this operation?						1 (1/0)											2 (1/1)						3 (2/1)
13. Which execution path is being taken in this case?		1 (0/1)	1 (1/0)		1 (1/0)	2 (2/0)	2 (1/1)	2 (1/1)	6 (5/1)		2 (1/1)	6 (4/2)	7 (7/0)	2 (2/0)	3 (2/1)			2 (2/0)		5 (2/3)	3 (3/0)	4 (4/0)	5 (4/1)
14. Under what circumstances is this method called or exception thrown?	1 (0/1)	1 (1/0)		1 (1/0)	3 (3/0)	2 (2/0)	2 (2/0)	1 (0/1)				2 (2/0)	1 (0/1)	4 (3/1)						2 (2/0)			20 (16/4)
15. In what order are these functions executed?	1 (1/0)				1 (1/0)	1 (1/0)						2 (2/1)											6 (5/1)
Total in the category																							
	7 (1/6)	7 (4/3)	9 (9/0)	3 (2/1)	20 (20/0)	14 (13/1)	8 (7/1)	6 (4/2)	11 (9/2)	1 (1/0)	5 (4/1)	12 (9/3)	12 (11/1)	3 (3/0)	9 (7/2)	1 (1/0)	4 (3/1)	2 (2/0)	11 (8/3)	3 (3/0)	7 (7/0)	8 (7/1)	163 (135/28)
Discovering Memory Usage at a Single Point of Time																							
16. How much memory does the execution of the entity (function, instance, line of code) allocate?	3 (3/0)		2 (1/1)		5 (4/1)	1 (1/0)	2 (2/0)	14 (14/0)	3 (3/0)	2 (2/0)	4 (4/0)	1 (0/1)	3 (1/2)	1 (0/1)	3 (2/1)				3 (2/1)	1 (1/0)		2 (2/0)	46 (40/6)
17. How much memory do these parts of the code allocate together?	3 (2/1)	3 (1/2)	4 (4/0)	4 (3)	1 (0/1)	5 (2/3)	5 (2/3)	3 (1/2)	1 (1/0)		1 (0/1)	2 (1/0)	4 (1/3)	1 (0/1)	1 (0/1)	1 (0/1)	2 (1/1)	3 (3/0)	5 (1/4)	1 (1/0)	1 (1/0)	1 (1/0)	46 (23/23)
18. How much memory in total is being allocated?			3 (2/1)		2 (2/0)	1 (0/1)	1 (0/1)		2 (0/2)														14 (5/9)
19. Why do these parts of code allocate this amount of memory?	1 (0/1)	2 (1/1)	4 (4/0)	3 (2/1)	3 (3/1)	1 (1/0)	1 (1/0)	5 (3/2)	6 (3/3)	1 (1/0)		2 (0/2)		1 (0/1)	1 (1/0)	1 (0/1)	1 (1/0)	1 (1/0)	17 (4/13)	3 (0/3)		1 (0/1)	55 (25/30)
20. How does this data structure look at runtime?						1 (1/0)		3 (2/1)															4 (3/1)
Total in the category																							
	6 (5/1)	4 (1/3)	11 (8/3)	4 (4/0)	9 (6/3)	9 (8/3)	15 (5/4)	16 (15/0)	9 (9/7)	5 (6/3)	5 (5/0)	3 (1/2)	9 (3/6)	4 (1/3)	3 (1/2)	2 (1/1)	5 (1/4)	7 (6/1)	23 (6/17)	4 (1/3)	4 (3/1)	2 (0/2)	165 (96/69)
Comparing and Contrasting Memory Usage																							
21. How much is the maximum memory peak?								1 (1/0)		1 (0/1)					1 (0/1)								3 (1/2)
22. How does memory usage evolve over time?			1 (1/0)			1 (0/1)		5 (3/2)				2 (0/2)		1 (0/1)		1 (0/1)			2 (0/2)		1 (0/1)		15 (5/10)
23. Which entities (functions, lines of code, instances) allocate most memory?	9 (6/3)	5 (3/2)	6 (4/2)	5 (4/1)	7 (2/5)	11 (5/6)	5 (1/4)	12 (8/4)	3 (0/3)	7 (4/3)	3 (1/2)	2 (1/1)	5 (4/1)	3 (0/3)	3 (1/2)	3 (2/5)	3 (4/2)	7 (4/2)	6 (4/10)	6 (3/1)	4 (3/4)	7 (3/4)	130 (62/68)
24. What will be the impact in memory behavior of this change?	1 (0/1)	1 (0/1)	3 (3/0)	1 (1/0)	2 (0/2)	1 (1/0)	3 (1/2)	1 (1/0)	1 (1/0)		1 (0/1)		2 (0/2)		1 (1/0)				2 (0/2)	8 (3/5)	3 (1/2)		30 (12/18)
25. What is the difference in memory behavior between these similar parts of the code (e.g., between sets of methods)?	1 (1/0)					1 (1/0)									1 (1/0)						1 (0/1)		4 (3/1)
26. What are the differences in memory behavior between this point of time and that point of time?			1 (0/1)		2 (0/2)	2 (0/1)	1 (0/1)				1 (0/1)		1 (0/1)		1 (0/2)					10 (0/10)		1 (0/1)	20 (0/20)
27. What is the difference in memory behavior between these code executions?		2 (1/1)	1 (0/1)			1 (1/1)			1 (1/0)	2 (0/2)					1 (0/1)								3 (3/0)
28. What part (function, line of code, instance) of this entity allocates the most memory?	1 (1/0)	1 (1/0)				2 (2/0)	3 (3/0)	2 (2/0)		1 (1/0)	1 (1/0)	1 (1/0)	2 (2/0)	1 (1/0)	2 (2/0)	1 (1/0)			6 (6/0)	2 (2/0)	3 (3/0)	1 (1/0)	30 (30/0)
Total in the category																							
	10 (7/3)	10 (6/4)	10 (5/5)	8 (7/1)	10 (3/7)	18 (7/11)	9 (4/5)	15 (13/6)	9 (8/5)	7 (4/5)	7 (2/5)	12 (2/1)	5 (5/7)	9 (2/3)	6 (4/5)	9 (3/3)	8 (3/6)	4 (4/4)	42 (13/29)	9 (6/3)	11 (6/5)	5 (2/3)	242 (116/126)
Discovering Memory Events																							
29. Where in the code are memory allocations made in this function?	6 (6/0)		2 (2/0)		1 (1/0)	1 (1/0)	1 (1/0)		4 (4/0)		7 (7/0)			1 (0/1)					2 (1/1)		1 (0/1)		26 (23/3)
30. Where are instances of this class created?								1 (1/0)					1 (1/0)										2 (2/0)
31. When are these objects garbage collected?						4 (3/1)		2 (1/1)				1 (1/0)									1 (1/0)		8 (6/2)
32. What data is being modified in this code?				1 (1/0)	2 (2/0)	1 (1/0)		1 (1/0)				1 (1/0)			1 (1/0)								7 (7/0)
33. What objects are allocated?						2 (2/0)																	2 (2/0)
34. Where is this variable or data structure being accessed?						1 (1/0)						1 (1/0)											2 (2/0)
Total in the category																							
	6 (6/0)	1 (1/0)	0 (0/0)	3 (3/0)	9 (8/1)	2 (2/0)	1 (1/0)	1 (1/0)	3 (3/0)	2 (1/1)	0 (0/0)	8 (7/1)	0 (0/0)	0 (0/0)	2 (0/0)	0 (0/0)	0 (0/0)	1 (1/0)	7 (7/0)	1 (0/1)	0 (0/0)	0 (0/0)	47 (42/5)
Total	41 (23/18)	30 (13/17)	38 (29/9)	25 (22/3)	50 (35/15)	61 (42/19)	34 (23/11)	48 (39/9)	59 (40/19)	22 (13/9)	25 (19/6)	32 (25/7)	48 (33/15)	16 (9/7)	30 (20/10)	11 (7/4)	23 (11/12)	23 (17/6)	91 (31/60)	17 (11/6)	33 (25/8)	18 (12/6)	775 (499/276)

when or in which situations some functions are called) belong to this category. This category contains nine questions, and 163 question occurrences (21%) were asked during work sessions. The most frequent questions in this category are: "13. Which execution path is being taken in this case?", "12. How many times is this entity (function or line of code) executed?", and "10. Which functions are called by this function?"

Similar to the first category, five of the eight questions in this group have been reported in previous studies [22], [23], [24], [52], [27]. Participants frequently asked questions

from this category when (i) understanding the execution of specific code that affected memory usage (allocations, releases, accesses) and (ii) identifying the root cause of a memory issue.

Comprehending code execution. To better understand why particular code parts (specific entities or allocation sites) were executed, participants often investigated the circumstances that caused their execution and the relationships between functions/methods. For instance, P12 identified a

function that allocated most memory and asked “*Under what circumstances is this `__new__` function called?*” P12 explored the relationships between functions and discovered that `__new__` function was called several times to generate multiple data frames that later are transformed into arrays using the numpy package.

Detecting the root cause of an issue. Programmers also centered on the control flow to detect the parts of code responsible for excessive or inefficient memory usage. For example, P19 located code parts that threatened memory consumption due to a suspicious memory increment and asked “*Which execution path is being taken to allocate these objects?*” to trace the root cause of the issue and analyze if the current code could be modified.

5.3 Discovering Memory Usage at a Single Point of Time

This category involves questions about discovering memory usage at a single point in time. We detected five questions and 165 question occurrences (21%) in this category. The most frequent questions were: “19. *Why do these parts of code allocate this amount of memory?*”, “16. *How much memory does the execution of the entity (function, instance, line of code) allocate?*”, and “17. *How much memory do these parts of the code allocate together?*”

In contrast to previous categories, only one question (question 20) was reported by previous studies [22], [23], [24], [52]. Therefore, seven questions in this category were not formally reported in the state of the art and are suitable for memory behavior analysis. Participants asked these questions to investigate memory behavior at a specific time (e.g., allocations made, memory occupied), a memory activity analysis considered relevant by other studies [15], [26], [31]. Answering these questions helps programmers understand how certain entities impact memory behavior.

Participants usually asked for the memory behavior of certain entities during the application’s execution. For example, P8 asked “*How much memory do the functions in charge of plotting my figures consume?*” to verify and confirm that these functions allocated most of the memory. Also, participants asked “19. *Why do these parts of code allocate this amount of memory?*” especially when they found (i) unexpected entities responsible for abnormal memory behavior or (ii) code that allocated more or less memory than expected. Some participants also look for information about the total memory allocated at a particular time. For instance, P3 asked “*How much memory in total was allocated so far?*” to analyze the impact on memory allocated by particular code parts during the execution. In addition, two participants asked “20. *How does this data structure look at runtime?*” when exploring the reason behind memory behavior at a certain point in more detail.

5.4 Comparing and Contrasting Memory Usage

Questions about comparing and contrasting memory usage belong to this category. We identified seven questions and 242 question occurrences (31%) grouped in this category. Question “23. *Which entities (functions, lines of code, instances) allocate most memory?*” is the most frequently raised by participants.

While a number of studies consider inspecting memory usage over time as an important memory activity [5], [15], [26], past studies have not formally reported the questions in this category. During our study, participants frequently raised questions from this category when they (i) investigated memory over time and (ii) compared memory usage between software versions or different code executions.

Investigating memory over time. Participants focused on how memory consumption varies during program execution. The latter helps programmers detect abnormal memory growth and potential leaks. For example, participants asked questions to detect anomalies when analyzing memory behavior over time: “21. *How much is the maximum memory peak?*”, “22. *How does memory usage evolve over time?*”, and “26. *What are the differences in memory behavior between this point of time and that point of time?*”.

Contrasting memory usage. Participants compared the memory behavior of entities to determine which are responsible for suspicious memory usage; thus, programmers asked “23. *Which entities (functions, lines of code, instances) allocate the most memory?*” and “28. *What part (function, line of code, instance) of this entity allocates the most memory?*”. Some participants tried to find opportunities to reduce memory usage by analyzing the memory allocated by parts of the code (“25. *What is the difference in memory behavior between these similar parts of the code (e.g., between sets of methods)?*”) or distinct code executions (“27. *What is the difference in memory behavior between these code executions?*”). As a result, programmers usually need to execute the code multiple times and compare the memory usage.

Furthermore, we noticed that when programmers propose a change, they often make assumptions or want to determine how it affects memory consumption and functionality (“24. *What will be the impact on memory behavior of this change?*”). For example, P19 mentioned “*I’m using a deep copy when I don’t really need it; I think I could change this part and reduce memory consumption quite a bit*”, and P8 said “*In this piece of code I could remove these operations or move them so that they are done at the end, only once, and optimize memory*”.

5.5 Discovering Memory Events

Questions about discovering and locating memory events are grouped in this category. A total of seven questions and 47 question occurrences (6%) were raised by participants and belong to this category. The most frequent question is “29. *Where in the code are memory allocations made in this function?*”

Previous studies [22], [23], [24], [52] about information needs reported three of the six questions in this category. Participants asked questions from this category to detect an anomaly and optimization opportunity by exploring (i) memory allocations, (ii) memory accesses, and (iii) memory releases.

Exploring memory allocations. Some participants inspected the data allocated in memory and the code responsible for this action. Consequently, participants asked: (“29. *Where in the code are memory allocations made in this function?*”, “30. *Where are instances of this class created?*”, “33. *What objects are allocated?*”)

Exploring memory accesses. Since the data allocated is used in different operations (read and write). Some participants

explored how the data allocated in memory was used and why. As a result, participants asked: “32. *What data is being modified in this code?*” and “34. *Where is this variable or data structure being accessed?*”

Exploring memory releases. Some participants asked “31. *When are these objects garbage collected?*” to examine if memory occupied by data that is no longer needed was released.

RQ1: What questions do Python programmers ask during memory behavior analysis? We identified 34 questions that programmers pose during memory behavior analysis. These questions include grasping source code and control flow to locate memory anomalies and potential improvements. Participants also explored memory behavior at specific times, anomalies over time, and comparisons between code versions. They delved into memory allocations, accesses, and releases. Notably, 14 of these questions align with those from previous studies, particularly in source code and control flow understanding. In total, our study introduces 20 distinct questions for memory analysis.

6 RQ2: HOW DO PYTHON PROGRAMMERS ANSWER THESE QUESTIONS USING MEMORY PROFILER TOOLS?

We detected that 665 question occurrences (86%) raised by participants were answered using features from Vismep and Tracemalloc. Note that we only consider a question as answered if a participant responded to this question only using the memory analysis tool's features, not the IDE itself. Table 5 illustrates the features of Tracemalloc and Vismep that participants used to answer the questions per category. The following subsections describe in more detail the actions performed to answer the questions.

6.1 Understanding Source Code

Participants responded to 150 question occurrences (95% of the occurrences in category) about understanding static structure and implementation.

Responding to question 1 (“*Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?*”) is about finding code parts to a particular functionality. To answer this question using Vismep, participants usually searched for a specific entity based on its name or structural component (file or module). Participants often then check the entity's code to ensure it is related to specific behavior and explore their relationships with other entities that could be involved in the same task. Consequently, participants often employed the *Call graph view* and the *Source code view*. For Tracemalloc, a participant manually inspected the source code of allocation sites based on the static information about the file and line number reported with *Display TOP* to verify if some sites were involved with the functionality that he considered problematic.

Questions 2 “*Which entities (functions, lines of code, class) belong to this file or module?*” and 3 “*Is there an entity named something like this in that unit (project, package, or*

class)?” require finding entities based on their structural component (file or module they belong to) or name. To answer these questions with Vismep, participants often (i) manually inspect the name and file of entities in diverse views (*Call graph view*, *Scatter plot view*, *Sub call graph view*), (ii) use the search mechanism to locate an entity with a particular name, and (iii) explore the *Source code view* of entities to ensure that they belong to a specific unit. When participants employed Tracemalloc, they answered question 2 by manually searching the names of files or modules in the textual reports (*Display TOP*, *Get traceback*) and exploring their source code.

Questions 4 “*What does the declaration or definition of this look like?*” and 5 “*What are the parts of this entity (function, instance, type)?*” involve inspecting the source code of entities. Participants responded to these questions by inspecting the source code of a given entity using *Source code view* from Vismep. Also, participants explored the source code of entities using the static information shown in diverse textual reports (*Display TOP*, *Compute differences*, *Get traceback*).

Responding to question 6 “*What is the behavior that these entities (functions, lines of code, instances) provide together?*” requires understanding the task in which specific parts of code were associated. Participants often answer this question by exploring the source code of entities and their relationships to understand how these entities impact the program's functionality. When participants used Vismep, they focused on inspecting *Source code view* and exploring the relationships using *Call graph view* or *Sub call graph view*. In the case of Tracemalloc, participants often opened and moved between several windows to inspect the source code of the targeted entities and their relationship.

6.2 Understanding Control Flow

A total of 137 question occurrences (84% of the occurrences in the category) were answered during sessions.

Questions 7 “*Which entities (functions, lines of code) are the most executed?*”, 11 “*How many times is this entity (function or line of code) executed?*” and 12 “*How many recursive calls happen during this operation?*” is about how entities were executed. To respond to these questions, participants used Vismep to examine the information about the entity's execution with the popup windows or visual hints. For instance, participants often used *Scatter plot view* to investigate the position of entities since the X-axis in the chart represents the number of executions. Participants also selected *Call graph view* or *Sub call graph view* to inspect the height of nodes (proportional to the number of executions) and look for nodes with loops among their edges to identify recursion. Participants could not respond to questions 11 and 12 using Tracemalloc.

Answering questions 8 “*Where is this method called or referenced?*” and 10 “*Which functions are called by this function?*” require examining control flow information, specifically the relationship between functions. When participants used Vismep, they responded to these questions by exploring the relationships between nodes in *Call graph view* and *Sub call graph view*. Sometimes they also inspected the code of certain functions/methods to indicate the line of code responsible for calling a function (*Source code view*). Participants could not answer question 8 employing Tracemalloc.

TABLE 5

Used features from Vismep and Tracemalloc to answer questions per category. Each column corresponding to a category represents the number of answered questions of this category using a feature.

Category	Feature	Understanding Source Code	Understanding Control Flow	Discovering Memory Usage at a Single Point of Time	Comparing and Contrasting Memory Usage	Discovering Memory Events	Total
Tracemalloc	Display top	39	1	38	65	2	145
	Compute differences	2	1	12	31	2	48
	Get traceback	5	8	5	6	0	24
	Get traced	0	0	2	3	0	5
	Combined reports	1	1	1	5	0	8
Vismep	Call graph view	45	47	43	22	2	159
	Source code view	18	2	7	27	18	72
	Scatter plot view	1	8	8	10	0	27
	Sub call graph view	4	29	6	10	0	49
	Combined views	35	40	21	21	11	128
Total		150	137	143	200	35	665

Questions 9 “When during the execution is this method called?”, 13 “Which execution path is being taken in this case?”, 14 “Under what circumstances is this method called or exception thrown?” and 15 “In what order are these functions executed?” consider understanding dynamic aspects of the control flow or data flow in a particular context. To answer questions 13, 14, and 15 using Vismep, participants investigated and navigated iteratively over the (i) code of functions (*Source code view*) and (ii) relationships between functions (*Call graph view*, *Sub call graph view*). When participants employed Tracemalloc, they answered these questions by (i) analyzing the chain of executions that lead to a particular memory allocation (*Get traceback*) and (ii) searching manually for the references to functions/methods and inspecting the respective code.

6.3 Discovering Memory Usage at a Single Point of Time

Participants responded to 143 question occurrences (87% of the occurrences in the category) about discovering memory usage at a point in time.

Responding to question 16 “How much memory does the execution of the entity (function, instance, line of code) allocate?”, and 17, “How much memory do these parts of the code allocate together?” require exploring the memory usage of one or multiple entities altogether. When participants answered these questions with Vismep, they often investigated the information about the memory usage of entities in the views (*Call graph view*, *Scatter plot view*, *Sub call graph view*). For Tracemalloc, participants used a number of function calls to report the memory usage of lines of code (*Display TOP*, *Compute differences*). Besides, with both tools, participants performed mental operations with the information from each group entity to respond to question 17.

Questions 18 “How much memory in total is being allocated?” is about exploring the memory allocated at a point. Some participants answered question 18 with Vismep by locating the root function responsible for the program execution and inspecting its memory behavior in *Call graph view* and *Scatter plot view*.

Question 19 “Why do these parts of code allocate this amount of memory?” considers dynamic and static aspects of the program to understand the reasons behind memory usage. To answer this question, participants usually focused on code parts and explored (i) their memory usage, (ii) other

code parts associated with their execution (control flow), and (iii) its source code to gain a better comprehension of program behavior. Consequently, when using Vismep, participants usually inspected the memory usage information, the relationships between functions (*Call graph view* or *Sub call graph view*), and the code (*Source code view*). For Tracemalloc, some participants often reported the memory used by parts of code and inspected in depth its source code. Other participants reported the changes in memory usage after and before executing these parts of code (*Compute differences* along with the chain of code execution that caused the memory allocations associated with them (*Get traceback*).

Question 20 “How does this data structure look at runtime?” is about inspecting the state of a particular data structure in a specific point in time; participants were unable to answer question 20 with either tool.

6.4 Comparing and Contrasting Memory Usage

A total of 200 question occurrences (83% of the occurrences in the category) were answered by participants.

Regarding question 21 “How much is the maximum memory peak?”, participants could not respond to this question with Vismep. In the case of Tracemalloc, participants selected functions to show the total memory usage (*Display TOP*, *Get traced memory*) and the memory peak (*Get traced memory*) in the textual reports.

Responding to question 22 “How does memory usage evolve over time?” and 26 “What are the differences in memory behavior between this point of time and that point of time?” considers understanding changes in memory usage over time. When participants used Tracemalloc, they answered question 22 by answering multiple occurrences of question 26. They often report if the memory increased or decreased after and before executing a function using the *Compute differences* feature. Consequently, some participants obtained information about the changes in memory usage over time by inspecting several reports from *Compute differences*. Some participants did not respond to question 22 using Vismep, and question 26 was not asked when Vismep was used.

To answer questions 23 “Which entities (functions, lines of code, instances) allocate most memory?”, 25 “What is the difference in memory behavior between these similar parts of the code (e.g., between sets of methods)?” and 28 “What part (function, line of

code, instance) of this entity allocates the most memory?”, participants compared the memory allocated by entities and located the code responsible for allocating most memory. For these cases, participants often explored and manually compared the visual cues of elements in diverse views of Vismep. For example, to identify functions responsible for allocating most memory, they searched the widest nodes in *Call graph view* or the points located at the top in *Scatter plot view* since the width and the position in Y-axis indicate the memory usage. To identify code lines or instances that allocate most memory inside a function, participants inspected the *Source code view* and located the lines with a darker background. For Tracemalloc, participants responded to these questions by reporting with *Display TOP* the sites that allocate the most memory along with the file and the line of number. In the case of question 25, participants compared the memory usage of code parts utilizing information from background color in *Source code view* using Vismep or checking changes using *Compute differences* with Tracemalloc.

Answering questions 24 “What will be the impact in memory behavior of this change?” and 27 “What is the difference in memory behavior between these code executions?” require to detect changes in memory usage between versions or executions. To respond to these questions using Tracemalloc, participants usually reported the memory usage during program execution through various features (*Display TOP*, *Get traced memory*) for each version or execution. Then, participants manually compared the information from these reports to locate changes in memory usage based on a change or a different input. Participants could not answer these questions with Vismep.

6.5 Discovering Memory Events

A total of 35 question occurrences (75% of the occurrences in the category) about discovering memory events were answered by participants.

Questions 29 “Where in the code are memory allocations made in this function?” is about locating the code responsible for any memory allocation inside a function. Participants answered question 29 by detecting code lines with a colored background in *Source code view* with Vismep and inspecting the code lines that allocate memory in reports (*Display TOP*, *Compute differences*) of Tracemalloc.

Answering questions 30 “Where are instances of this class created?”, 31 “When are these instances garbage collected?” and 33 “What objects are created?” consider understanding information about the creation and release from the memory of instances. Participants could not answer question 30 using Vismep, and occurrences from this question did not arise when participants used Tracemalloc. To answer question 31, a participant detected and inspected the points associated with certain instances and when the memory decreased with *Compute differences* from Tracemalloc. Participants could not respond to questions 31 and 33 using Vismep, and no occurrence from question 33 was asked by participants when using Tracemalloc.

Question 32 “What data is being modified in this code?” and 34 “Where is this variable or data structure being accessed?” are about exploring the state of a particular data structure or locating the code responsible for accessing it. To answer questions 32 and 34 with Vismep, participants often examined the

source code associated with a particular data and explored their relationships with other entities using *Source code view* and *Call graph view*. Occurrences from questions 32 and 34 were not asked when participants used Tracemalloc.

RQ2: How do Python programmers use memory profiler tools? Python programmers use Vismep, especially its *Call graph view* and *Source code view*, along with combined views, and Tracemalloc, particularly with the *Display top* and *Get Traceback* features to understand source code and control flow. For questions about memory behavior at a single point in time and its evolution, they rely on Vismep’s *Call graph view* and other combined views, as well as Tracemalloc’s *Display top* and *Compute differences* features. To answer questions regarding contrasting memory behavior between versions and executions, participants often manually compare multiple reports. Additionally, Vismep’s *Source code view* and combined views, along with Tracemalloc’s *Display TOP* and *Compute differences*, partially support questions about memory allocations and releases.

7 RQ3: WHICH TYPE OF QUESTIONS DID PYTHON PROGRAMMERS NOT ANSWER, AND WHAT BARRIERS DID THEY FACE?

A total of 110 question occurrences (14%), as posed by participants, either (i) cannot be addressed using the functionalities provided by Vismep or Tracemalloc, or (ii) practitioners are unable to derive any benefit from the memory tool features. Table 6 summarizes the unanswered questions by category, along with possible reasons why these questions remained unanswered. Subsequent subsections will offer more detailed insights into participants’ efforts and challenges in addressing these questions, along with recommendations for enhancing the design of memory analysis tools.

7.1 Understanding Source Code

A total of 8 question occurrences (5%) about understanding source code were not answered.

A number of participants did not answer question 2 “Which entities (functions, lines of code, class) belong to this file or module?” using Vismep because after manually exploring several entities, they gave up due to the expensive inspection and mental fatigue. They manually searched since Vismep does not allow filtering entities considering the structural component. We notice that sometimes a manual search was also performed for question 3 “Is there an entity named something like this in that unit (project, package, or class)?”; however, these participants were able to answer this question because the search was limited to finding one entity and not groups of entities.

Regardless of the tool, participants inspected relationships between several entities to better understand the group’s functionality. However, some failed to respond to question 6 “What is the behavior that these entities (functions, lines of code, instances) provide together?” because the operations performed required considerable mental effort.

TABLE 6
Detected challenges that participants faced when trying to answer questions per category.

Category	#Unanswered Questions	Challenges
Understanding Source Code	8	Filtering entities per structural component. Inspecting relationships between groups of entities.
Understanding Control Flow	26	Fine-grained exploration of entities execution
Discovering Memory Usage at a Single Point of Time	22	Fine-grained exploration of entities memory usage. Connecting static and dynamic information.
Comparing and Contrasting Memory Usage	42	Unadequate reports for memory evolution. Contrasting memory behavior between code parts, versions, executions.
Discovering Memory Events	12	Insufficient information about allocations, releases and accesses.

7.2 Understanding Control Flow

Participants did not respond to 26 question occurrences (16%) related to understanding control flow.

Participants did not answer questions 9 “*When during the execution is this method called?*” regardless of the tool because it involves inspecting the context of a particular point, and the tools used do not extract this information. Some participants chose not to respond to questions 13 “*Which execution path is being taken in this case?*”, 14 “*Under what circumstances is this method called or exception thrown?*” and 15 “*In what order are these functions executed?*” The latter occurs because participants considered that tools do not report detailed information about control or data flow in a particular context (e.g., considering cases or circumstances, execution order). Therefore, participants could not point out in which context certain behavior occurs. Additionally, some participants did not answer questions 8 “*Where is this method called or referenced?*”, 11 “*How many times is this entity (function or line of code) executed?*” and 12 “*How many recursive calls happen during this operation?*” using Tracemalloc because it lacks features to extract this information.

7.3 Discovering Memory Usage at a Single Point of Time

A total of 22 question occurrences (13%) about discovering memory usage at a single point were unanswered.

One participant was unable to respond to a variation of question 16 “*How much memory does this variable allocate?*” using Tracemalloc because he could not find a suitable feature to inspect variables at that level. Furthermore, some participants did not answer question 17 “*How much memory do these parts of the code allocate together?*” when analyzing multiple lines of code in Vismep. This stemmed from the fact that Vismep does not provide a textual breakdown of memory usage for each line of code but instead offers visual cues within the background code. Consequently, participants believed that more specific information was needed to address this question for groups of code lines.

Participants sometimes could not answer question 19 “*Why do these parts of code allocate this amount of memory?*” because the navigation between dynamic information (memory usage, program execution) and static information (source code, structural component) was considered inadequate. Usually, participants found suspicious memory behavior but had difficulties better understanding the implementation

or the circumstances in which specific behavior occurs due to the challenges mentioned in understanding the source code and control flow. Additionally, question 20 “*How does this data structure look at runtime?*” remained unanswered because neither of the tools allowed users to inspect a data structure at a specific moment in time.

7.4 Comparing and Contrasting Memory Usage

Participants did not respond to 42 question occurrences (17%) about comparing and contrasting memory usage.

Questions 21 “*How much is the maximum memory peak?*” and 22 “*How does memory usage evolve over time?*” were not answered using Vismep. This situation occurs because Vismep summarizes the memory footprint using a call graph view instead of a more familiar visualization, such as a time-series chart. Besides, participants also faced difficulties using Tracemalloc to answer question 22. The latter occurs because participants must manually compare the reports for each snapshot taken. In some cases, they give up answering the question due to the high mental demand involved.

Some participants did not answer questions 23 “*Which entities (functions, lines of code, instances) allocate most memory?*” and question 25 “*What is the difference in memory behavior between these similar parts of the code (e.g., between sets of methods)?*” using Vismep because it does not textually report information about the memory behavior of a particular instance. Participants often encountered difficulties manually comparing the memory behavior in multiple reports with both tools to answer questions 24 “*What will be the impact in memory behavior of this change?*” and 27 “*What is the difference in memory behavior between these code executions?*”

7.5 Discovering Memory Events

A total of 12 question occurrences (25%) about discovering memory events were not answered.

Participants failed to answer question 30 “*Where are instances of this class created?*” with Vismep because it does not explicitly report this information as the classic list of instances (see Section 2.1). Participants struggled to manually identify functions responsible for creating specific instances because it took time and effort. Most participants also did not answer questions 31 “*When are these instances garbage collected?*” and 33 “*What objects are created?*” using Vismep and Tracemalloc because they only show the amount of

memory allocated or released but do not reveal the objects involved. Furthermore, both tools do not provide enough information to answer question 34 “*Where is this variable or data structure being accessed?*”; thus, participants need more support to answer this question.

RQ3: Which type of questions did Python programmers not answer, and what barriers did they face?

Participants encountered difficulties in addressing certain types of questions, such as those involving filtering entities based on structural components, examining relationships among groups of entities, exploring fine-grained information (e.g., context during execution, detailed memory events) about entities, and contrasting memory behavior between different sections of code, versions, or executions. As a result, participants identified several issues, including a lack of fine-grained information, limited interaction options (e.g., filtering and selection), inadequate reporting (e.g., connecting static and dynamic information), and the absence of essential support features (e.g., contrasting memory behavior).

7.6 Recommendations

We present four recommendations based on the study results and the participants’ answers to the post-study questionnaire. In the following, we describe each of these recommendations.

Provide IDE integration. Standard features of IDEs support answering questions about source code and control flow (e.g., search entities by name, structural component) [23], [27]. These questions were helpful when identifying memory anomalies and detecting opportunities for improvement. For this reason, we encourage memory tool builders to provide integration with IDE to make an adequate connection between static and dynamic information to support practitioners when analyzing memory behavior [6], [26].

Carefully select report presentation and interactions. As mentioned above, how information is presented and the techniques used for navigation and exploration impact the comprehension of information [11], [12], [13], [14]. In this study, participants mentioned that exploring time-series charts when analyzing memory evolution is familiar. Participants also pointed out that this report presentation could reduce the mental operations to answer questions about memory evolution with Tracemalloc and Vismep. On the other hand, allowing metric selection could reduce the complications when comparing entities based on the default visual mapping [6], [15], [60].

Use interactions for fine-grained information. Extracting and reporting a vast amount of data during program execution could be tedious and overwhelming [15], [31]. For this reason, we suggest that memory tool builders report fine-grained information based on the developer’s demands and preferences for report presentation. For instance, most participants appreciated Vismep’s interaction mechanisms, which allowed them to navigate through several visualizations side by side, providing connected information. However, Vismep could be further enhanced by incorporating features

for entity selection and filtration (e.g., based on memory usage) and offering more detailed information on demand (e.g., memory events within specific code lines). This would enable users to focus on the most relevant elements [60].

Provide support for contrasting memory behavior. Our results show that participants needed to contrast memory behavior between different code sections, versions, and executions. While only a few memory analysis tools currently support these activities, we believe that these capabilities can significantly influence research in the field, particularly when analyzing the impact of a change on memory behavior [15]. Therefore, we recommend that tool developers provide support for comparing elements at various levels of memory behavior analysis.

8 DISCUSSION

We discuss our findings and contrast them with other prior work as necessary.

8.1 Questions Asked By Participants

We found three categories centered on memory usage: (i) discovering memory consumption at a single point in time, (ii) comparing and contrasting memory consumption, and (iii) discovering memory events. Although these findings may not be surprising since they are involved with primary activities (e.g., performance, inspection of memory anomalies) mentioned in previous work [26], [27], we provide empirical evidence about their relevance.

We also detected questions about understanding the source code and the control flow. We included these questions because participants asked them to answer questions from the rest of the categories. To illustrate, after locating allocation sites responsible for abnormal memory behavior, several participants asked “*Why do these parts of code consume this amount of memory?*” Then, to answer this question, participants inspected the source code of the particular allocation sites (“*What does the definition of this look like?*”) and explored the control flow to understand why that code is executed (“*Under what circumstances is this method called?*”). Our results show that programmers require support for software comprehension when addressing a memory issue or determining the root cause of a potential failure. For this reason, we believe that these questions would be asked regardless of the tools used.

Precision of questions. The questions reported were inferred based on the questions explicitly asked by the participants or their speech, along with the actions performed during work sessions. Although some questions may seem very general, they represent the programmer’s intentions. For instance, question “24. *What will be the impact on memory behavior of this change?*” was inferred considering different verbalized thoughts (see Section 5.4). Accordingly, a “change” could involve distinct elements; for P19 it is the elimination of a deep copy, and for P8 it is to avoid the repetitive execution of code lines occupying a large amount of memory.

We also note that programmers asked more global questions due to tool features, tool limitations (see Section 4.2 and Section 7), and the open questions given during the work sessions. Thus, the questions presented can include vague

and irrelevant elements. Based on the latter, we consider further exploration necessary to distinguish the more precise and essential entities. For example, conducting a study with a specific software application with particular memory bugs.

Questions of other studies. As mentioned in Section 5, we reported questions that were and were not reported in studies on developer information needs. Most questions about discovering memory usage at a single point in time and comparing and contrasting memory usage were not explicitly identified by any other study. This situation may occur due to the lack of information needs during memory behavior analysis [15]. Consequently, our study provides valuable information to guide researchers in designing tools that adequately support programmers in this context.

Furthermore, most questions about understanding source code and control flow were reported by literature focused on programming change tasks [22], [23], [24], [52]. Similarly, questions about discovering memory events were classified as questions related to data flow or performance in previous studies [22], [27]. These situations illustrate that programmers asked common questions about software comprehension, whether to perform programming change tasks, fix memory issues or determine the cause of a failure.

Questions not raised but expected. Although we provide a catalog of questions to illustrate the needs of programmers during the analysis of memory behavior, some questions could be missing. Several studies mentioned the relevance of supporting memory anomaly detection (e.g., memory bloats, memory leaks, unusual garbage collector behavior) [26], [15], [21], [25]. Consequently, questions like “Which objects are alive until this time?”, “Which objects are kept alive by a given object/variable?”, “Which object did not survive during the garbage collection?” would be expected. Even more technical questions about garbage collection information, threading, and the memory occupied by Python vs. native code, among others, could arise [20], [50], [51]. Note that these questions could be missing in our study due to tool limitations, participants’ backgrounds, and characteristics of the applications under study. Thus, further exploration is necessary to identify more questions and complete our catalog.

Question frequency. We divided our participants into two groups, G1 and G2. Participants from G1 first used Vismep (FP) and then Tracemalloc (SP), while in G2, the order of tools was reversed. We observed that participants from G1 usually ask more questions than participants from G2. We also noticed that questions about discovering memory events were asked more often in G1 than in G2. One explanation for this difference is the applications selected by participants and how they address issues. For example, some participants (P6, P9) from G1 chose applications with memory anomalies (leaks, bloats) and centered on exploring the memory allocations and releases.

We also detected that questions from the first, second, and last categories arose more frequently when participants used Vismep. One reason for this difference is that programmers considered that Vismep gives more support to answer these questions than Tracemalloc; thus, more questions arose. In addition, questions from the third and fourth categories were asked more often during the first phase of both groups. One contributing factor may be the learning effect in the

study. In other words, some participants asked questions about the program’s memory usage during the second phase considering the knowledge acquired from the first phase. For instance, some participants could avoid asking about the memory used by some entities they know are not a threat to memory usage based on analysis from the first phase.

Learning effect. Participants analyzed the same application with Vismep and Tracemalloc. The order for using each tool was defined based on the participant’s group (G1 and G2). Each participant decided when to end up the first phase to start the second phase and use another tool. Therefore, we did not force participants to switch from one tool to another. In the second phase, participants generally (i) located or confirmed the information or assumptions they had in the first phase, (ii) used the knowledge from the first phase to analyze the code further, and (iii) detected new information. We also noted that participants asked more or fewer questions between phases due to the tool’s limitations or because they had prior knowledge of the first phase. To illustrate, participants asked more questions about memory evolution over time using Tracemalloc due to the support that this tool offers. In addition, as mentioned before, we discussed that the frequency of questions might vary due to different reasons (e.g., tool usage, application selected). Note that our goal is not to compare Vismep and Tracemalloc or rate the questions for relevance based on the frequency.

Furthermore, we increased the diversity and range of the questions asked by practitioners due to dividing participants into two groups balanced based on their study field and observing how they analyzed the same application with both tools in a different order. We also found that programmers analyze memory usage and address memory issues using a wide range of features that are not necessarily supported by a single tool.

Study field effect. We examined if the questions asked may vary based on the participant’s study field. We analyzed the differences in questions between two groups of participants: (i) group C, which contains participants in the Computer Science field, and (ii) group N, which contains the rest of the participants. As a result, group N participants often asked more questions about understanding source code and control flow. In addition, group C participants usually asked more questions about discovering memory usage at a point in time and comparing memory usage. These differences could be that participants from fields distinct to Computer Science may need to explore several entities at different levels to understand the program’s behavior and structure.

Self-assessment expertise effect. We inspected if the questions asked may vary based on the participant’s expertise in Python. We analyzed the differences in questions between two groups of participants: (i) group E, which contains participants with self-assessment expertise above 3.2 (average), and (ii) group N, which contains the rest of the participants. As a result, group N participants usually asked more questions about understanding source code, control flow, and discovering memory events. Group E participants often asked more questions about discovering memory usage at a point in time and comparing memory usage. A factor that may cause these differences could be that participants considered themselves with expertise above

regular in Python could obtain information to modify their code, optimize memory usage and perform fewer operations to comprehend the program's functionality.

Students vs. professionals. We analyzed whether the questions may differ based on whether the participant is an undergraduate or a professional. As a result, regardless of whether they are professionals or not, participants often ask a similar frequency of questions about understanding control flow, discovering memory usage at a single point in time, and discovering memory events. In addition, professional participants usually asked more questions about understanding source code and comparing memory usage. This difference could occur because professional participants may require to inspect and compare several entities at different levels to comprehend their application's behavior and the impact of some elements in memory usage.

Gender effect. Some studies [61], [62] have shown differences between men and women when debugging. Our study involved twenty-two participants, of which six were women. We detected that both men and women ask questions from the five categories. We also found that women usually ask more questions about discovering memory events, while in the other categories, the question frequencies are often higher for men. However, we could not ensure that these situations are due to gender since our study presents (i) a small sample, which is not balanced between men and women, and (ii) variations in sessions (e.g., applications selected). Furthermore, any observations about the questions' occurrences should be treated carefully as the sessions from which the data is extracted varied in diverse ways (e.g., different applications, presence of memory issues). Nonetheless, studies of the gender-related issues within tasks related to memory consumption analysis would extend this work.

8.2 Answering Questions

We observed that Vismep and Tracemalloc could support programmers in answering most questions inferred in this study. However, Section 7 details that participants could not answer some questions. Consequently, we detected participants' challenges during the process and made four recommendations when designing memory analysis tools.

Support of other tools. We believe that most approaches in Python (see Table 1) and other programming languages [15], [26] could support most of the questions inferred in our study due to the reported data and activities they claim to support. To illustrate, some of the approaches of Python [50], [51] and memory analysis tools for other programming languages such as AntTracks [5], Yourkit [37], JProfiler [38], could provide enough support to answer most of the questions in the third and fourth categories by allowing analysis of the evolution of memory usage and inspecting memory at one point in time. On the other hand, other options [5], [46], [47], [48] may provide adequate support for answering some questions about memory events, specifically the creation and release of specific objects. However, it is worth noting that most Python analysis tools do not provide explicit information to support the latter category. Nonetheless, further research must be conducted with other tools to

confirm if programmers could use the information reported to answer these questions.

Furthermore, we consider that some approaches could not adequately support answering questions about source code and control flow because they do not adequately connect static and dynamic information. Consequently, we strongly suggest proper integration with IDEs, as our study, like previous literature [6], [26], highlights that it would facilitate the solution of memory issues.

Learning effect. We defined that a question is answered only if the participant demonstrates and explains how she/he employs the corresponding tool to respond to the question and mentions the answer. Vismep and Tracemalloc differed in the information reported and how information is provided and navigated. Consequently, programmers used different strategies to manage both tools and obtain certain information. For example, participants compared the visual hints of elements in the *Call graph view* and *Source code view* to identify allocation hot spots with Vismep. On the other hand, participants used the *Display TOP* from Tracemalloc to obtain the same information.

Study field effect. We observe that participants, regardless of their study field and tool's usage, usually answered a similar proportion of questions for all categories except understanding control flow. Computer science participants tend to answer more questions about control flow. The latter may be because these participants may be more accustomed to performing operations to explore information on this aspect.

Self-assessment expertise effect. We analyzed the differences in questions between two groups of participants: (i) group E, which contains participants with self-assessment expertise above 3.2 (average), and (ii) group N, which contains the rest of the participants. Participants from both groups, regardless of the tool's use, often answered a similar proportion of questions for all categories except discovering memory events. Participants from group E usually responded to more questions about discovering memory events. One factor may be that experienced participants could extract information by using different features to obtain information that was not explicitly displayed (e.g., allocations, releases).

Students vs. professionals. We detected that participants, regardless of whether they were professionals or not, often answered a similar proportion of questions for understanding source code and discovering memory events. Professional participants tended to answer more questions about discovering memory usage at a single point, and students usually responded to more questions about control flow and comparing memory usage. These differences could exist because professional participants may perform operations with a large amount of data, being difficult and complex to obtain the required information to answer the questions.

Gender effect. We noticed that participants, regardless of gender, often answered a similar proportion of questions for all categories except discovering memory usage at a single point and discovering memory events. Women tend to answer more questions about discovering memory usage at a single point and discovering memory events than men. This situation could occur because women were found efficient users of tinkering in previous studies [61], [62].

9 THREATS TO VALIDITY

Our study and results are subject to validity threats [63]. To carefully identify possible threats and analyze how their impact may be mitigated, we decided the following:

Conclusion validity. Our conclusions are founded on an exploratory study involving programmers analyzing the memory usage of their Python applications using memory profilers. However, our conclusions are based on observing only twenty-two participants, a relatively low sample. We try to reduce this threat by selecting participants with diverse backgrounds (e.g., study fields, experience in Python programming). Although we had no indication that increasing the number of participants may invalidate our result, the frequency and precision of questions in our results may be affected.

Internal validity. The second author performed the data collection and transcription of each work session. Then, the first author checked the generated spreadsheets based on the video recordings and event logs to minimize inconsistencies. Furthermore, the second author conducted the steps to identify the questions asked by the participants. Identifying concrete questions based on user behavior may be inaccurate, as participants did not explicitly verbalize their thoughts. Finally, the process of defining general questions may suffer from uncertainty. For instance, it can be challenging to distinguish question 23 "What will be the impact in memory behavior change?" from question 26 "What is the difference in memory behavior between these code executions?". To minimize inaccuracy in the inference process, the first author contrasted different scenarios and events related to the same questions and checked if the inferred questions were consistent with the information from spreadsheets. All the authors held two meetings to discuss any discrepancy in the inferred questions' consistency.

Regarding the classification scheme, the first author conducted a thematic analysis to organize the inferred questions based on the information needed and the programmer's behavior to answer a question. The codes and themes generated vary depending on the coder's experience, level of abstraction, and point of view. To mitigate this threat, two authors checked the consistency of the process by examining the description of themes with the associated data. We conducted two meetings to discuss and resolve the disagreements among generated codes and themes. Additionally, two reviewers independently categorized the inferred questions using the classification scheme, and a measure of agreement between reviewers was calculated to validate the reliability.

Construct validity. We focus on understanding how programmers analyze the memory usage of Python programs using memory profiler tools. Therefore, we voluntarily centered on the Python programming language. For the study, participants chose software applications with which they were familiar to analyze them during the sessions. Furthermore, data from each work session was carefully examined and collected using records, logs, and observation.

External validity. The selected memory profiler tools and the individual differences among participants influence the programmers' questions and how they answer those.

Consequently, the questions could vary given a completely distinct set of memory profiler tools, participants, or applications. The latter must be considered when interpreting or generalizing the results. We mitigated this threat by selecting programmers with different backgrounds and experience levels who chose applications under study. In addition, we opted for memory profiler tools that provide diverse information and report presentations commonly offered by other Python memory tools. We also discussed and analyzed the tool support for answering questions considering the features of other tools. As a result, we noticed that many inferred questions were independent of whether they could be answered with the tools. Furthermore, no new questions were detected in the last work sessions. The latter suggests that some of our results will likely generalize to other tools. The selection of Vismep and Tracemalloc enables us to identify various memory analysis questions related to memory increments, memory releases, bottlenecks, and their relationship with the source code. However, we plan to explore additional tools that further enhance the scope of our study.

Our results also need to be interpreted relative to the programming language and the open questions used in the study. We focused on understanding the impact of memory profiler tools on supporting programmers during memory consumption analysis for Python applications. The participants selected Python applications with which they were familiar and answered open questions to ensure that participants looked for the information they considered valuable. It is important to note that we did not extend our study to encompass other programming languages or a broader array of memory profiling tools due to the practical challenges associated with conducting such expansive research. For instance, data collection and transcription were resource-intensive, requiring a full day's work per session. Additionally, our study deliberately maintains a general focus without specifying highly detailed tasks, such as identifying five allocation hotspots. Participants were encouraged to exercise their judgment when selecting applications, and no restrictions were placed on project or system size or complexity. While our study provides valuable insights within these confines, there is room for future research to diversify and explore various programming languages and toolsets, thus enabling a more comprehensive understanding of memory analysis needs in specific contexts.

As mentioned above, the sessions in our study varied along several dimensions, and we have not thoroughly analyzed how the questions asked and the answering behavior varied along those dimensions. Although we discussed in Section 8 the differences between question frequencies and the questions asked and answers by participants with diverse backgrounds, this information is insufficient to conclude that the study field or the programming experience affects the questions asked and how the tools are used. Obtaining precise information about the latter would require a study set up with more controls on the dimensions along which the sessions are allowed to vary.

10 CONCLUSION

We conducted an observational study to provide (i) an empirically-based set of questions that programmers ask during memory usage analysis and (ii) a report about how programmers those questions with Vismep and Tracemalloc. In our exploratory study, we observed 22 programmers analyzing the memory consumption of Python applications with which they were familiar using these two tools. We found 34 different questions and organized them into five categories based on the information needed and the programmer's behavior: (i) understanding source code, (ii) understanding control flow, (iii) discovering the memory usage at a single point of time, (iv) comparing and contrasting memory usage, and (v) discovering memory events.

Vismep and Tracemalloc generally provide good support for answering most questions. However, participants often encountered difficulties due to (i) the lack of fine-grained information, (ii) omission of useful interaction options (e.g., filtering, selecting), (iii) inadequate reporting (e.g., failing to connect static and dynamic information), and (iv) missing support features (e.g., contrasting memory behavior). Consequently, our research offers four key recommendations to guide researchers and tool developers in creating, designing, and evaluating memory analysis tools. These recommendations encompass the integration with IDE, careful selection of report presentation and interactive mechanisms, utilization of interactions to provide fine-grained information, and providing support for contrasting memory behavior.

The significant implication is that there is still much to learn about how programmers analyze memory behavior. For this reason, we also discussed diverse aspects (e.g., precision of questions, support of other tools) and we point out that multiple studies beyond this are needed to fully understand how programmers perform these activities in different contexts. Those insights can be leveraged to create new tools or improve current tools to support programmers in their development environments.

ACKNOWLEDGE

Alison Fernandez Blanco is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PFCHA/Doctorado Nacional/2019-21191851. Alexandre Bergel is grateful to the ANID FONDECYT Regular project 1200067 for having partially sponsored the work presented in this article. Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciación Folio 11220885 for supporting this article.

REFERENCES

- [1] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, Dec 2014. [Online]. Available: <https://doi.org/10.1007/s10664-013-9258-8>
- [2] G. Xu and A. Rountev, "Detecting inefficiently-used containers to avoid bloat," *SIGPLAN Not.*, vol. 45, no. 6, p. 160–173, Jun 2010. [Online]. Available: <https://doi.org/10.1145/1809028.1806616>
- [3] N. Mitchell and G. Sevitsky, "The causes of bloat, the limits of health," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007, pp. 245–260.
- [4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, Jun 2005. [Online]. Available: <https://doi.org/10.1145/1064978.1065034>
- [5] M. Weninger, E. Gander, and H. Mössenböck, "Guided exploration: A method for guiding novice users in interactive memory monitoring tools," *Proc. ACM Hum.-Comput. Interact.*, vol. 5, no. EICS, may 2021. [Online]. Available: <https://doi.org/10.1145/3461731>
- [6] A. F. Blanco, J. P. S. Alcocer, and A. Bergel, "Effective visualization of object allocation sites," in *2018 IEEE Working Conference on Software Visualization (VISOFT)*. IEEE, 2018, pp. 43–53.
- [7] S. Byma and J. R. Larus, "Detailed heap profiling," *SIGPLAN Not.*, vol. 53, no. 5, p. 1–13, Jun 2018. [Online]. Available: <https://doi.org/10.1145/3299706.3210564>
- [8] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 151–160.
- [9] D. Lo, N. Nagappan, and T. Zimmermann, "How practitioners perceive the relevance of software engineering research," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, Aug 2015, pp. 415–425. [Online]. Available: <https://doi.org/10.1145/2786805.2786809>
- [10] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. ACM, 2010, p. 515. [Online]. Available: <https://doi.org/10.1145/1806799.1806874>
- [11] Y. Park and C. Jensen, "Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers," in *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2009, pp. 3–10. [Online]. Available: <http://ieeexplore.ieee.org/document/5336433/>
- [12] S. Diehl, "Software visualization," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 718–719.
- [13] S. Ducasse, M. Lanza, and R. Bertuli, "High-level polymetric views of condensed run-time information," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings*. IEEE, 2004, pp. 309–318. [Online]. Available: <http://ieeexplore.ieee.org/document/1281433/>
- [14] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211–266, 1993. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1045926X83710153>
- [15] A. F. Blanco, A. Bergel, and J. P. S. Alcocer, "Software visualizations to analyze memory consumption: A literature review," *ACM Comput. Surv.*, vol. 55, no. 1, Jan 2022. [Online]. Available: <https://doi.org/10.1145/3485134>
- [16] K. J. Millman and M. Aivazis, "Python for scientists and engineers," *Computing in Science Engineering*, vol. 13, no. 2, pp. 9–12, 2011. [Online]. Available: <http://ieeexplore.ieee.org/document/5725235/>
- [17] K. Srinath, "Python—the fastest growing programming language," *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 12, pp. 354–357, 2017.
- [18] "Tracemalloc - trace memory allocations," accessed: 2022-01-27. [Online]. Available: <https://docs.python.org/3/library/tracemalloc.html>
- [19] "vprof: Visual profiler for python," accessed: 2022-01-27. [Online]. Available: <https://github.com/nvdv/vprof>
- [20] "A python memory profiler for data processing and scientific computing applications," accessed: 2022-01-27. [Online]. Available: <https://github.com/pythonspeed/filprofiler>
- [21] A. F. Blanco, A. Bergel, J. P. S. Alcocer, and A. Q. Córdova, "Visualizing memory consumption with vismep," in *2022 Working Conference on Software Visualization (VISOFT)*, 2022, pp. 108–118.
- [22] J. Kubelka, R. Robbes, and A. Bergel, "Live programming and software evolution: Questions during a programming change task," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, May 2019, pp. 30–41. [Online]. Available: <https://ieeexplore.ieee.org/document/8813258/>
- [23] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, 2006, pp. 23–34. [Online]. Available: <https://doi.org/10.1145/1181775.1181779>

- [24] —, “Asking and answering questions during a programming change task,” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, Jul 2008. [Online]. Available: <http://ieeexplore.ieee.org/document/4497212/>
- [25] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak, “Memory and resource leak defects and their repairs in java projects,” *Empirical Software Engineering*, vol. 25, no. 1, pp. 678–718, Jan 2020. [Online]. Available: <https://doi.org/10.1007/s10664-019-09731-8>
- [26] M. Weninger, P. Grünbacher, E. Gander, and A. Schörghenhuber, “Evaluating an interactive memory analysis tool: Findings from a cognitive walkthrough and a user study,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. EICS, pp. 1–37, Jun 2020. [Online]. Available: <https://doi.org/10.1145/3394977>
- [27] T. D. LaToza and B. A. Myers, “Hard-to-answer questions about code,” in *Evaluation and Usability of Programming Languages and Tools*, 2010, pp. 1–6.
- [28] M. D. Bond and K. S. McKinley, “Tolerating memory leaks,” *SIGPLAN Not.*, vol. 43, no. 10, p. 109–126, Oct 2008. [Online]. Available: <https://doi.org/10.1145/1449955.1449774>
- [29] W. De Pauw and G. Sevitsky, “Visualizing reference patterns for solving memory leaks in java,” in *European Conference on Object-Oriented Programming*. Springer, 1999, pp. 116–134.
- [30] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O’Sullivan, T. Parsons, and J. Murphy, “Patterns of memory inefficiency,” in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 383–407. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2032497.2032523>
- [31] P. Gralka, C. Schulz, G. Reina, D. Weiskopf, and T. Ertl, “Visual exploration of memory traces and call stacks,” in *2017 IEEE Working Conference on Software Visualization (VISOFT)*. IEEE, Sep 2017, pp. 54–63. [Online]. Available: <http://ieeexplore.ieee.org/document/8091186/>
- [32] M. Weninger, L. Makor, E. Gander, and H. Mössenböck, “Anttracks trendviz: Configurable heap memory visualization over time,” in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA: ACM, 2019, p. 29–32. [Online]. Available: <https://doi.org/10.1145/3302541.3313100>
- [33] R. L. Veroy, N. P. Ricci, and S. Z. Guyer, “Visualizing the allocation and death of objects,” in *2013 First IEEE Working Conference on Software Visualization (VISOFT)*. IEEE, 2013, pp. 1–4.
- [34] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, *Visualizing the execution of Java programs*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2269, ch. chapter 12, pp. 151–162. [Online]. Available: http://link.springer.com/10.1007/3-540-45875-1_12
- [35] S. P. Reiss, “Visualizing the java heap to detect memory problems,” in *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, Sep 2009, pp. 73–80. [Online]. Available: <http://ieeexplore.ieee.org/document/5336418/>
- [36] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer, “Heapviz: interactive heap visualization for program understanding and debugging,” in *Proceedings of the 5th international symposium on Software visualization*. ACM Press, 2010, pp. 53–62. [Online]. Available: <https://doi.org/10.1145/1879211.1879222>
- [37] YourKit, “Yourkit java and .net profilers,” accessed: 2022-01-27. [Online]. Available: <https://www.yourkit.com/>
- [38] EJ-technologies, “Java profiler - jprofiler,” accessed: 2022-01-27. [Online]. Available: <https://www.ej-technologies.com/products/jprofiler/overview.html>
- [39] R. Wetzel, M. Lanza, and R. Robbes, “Software systems as cities: A controlled experiment,” in *Proceedings of the 33rd International Conference on Software Engineering*, May 2011, pp. 551–560. [Online]. Available: <https://doi.org/10.1145/1985793.1985868>
- [40] E. Tufté and P. Graves-Morris, “The visual display of quantitative information; 1983,” 2014.
- [41] S. P. Reiss and M. Renieris, “Jove: Java as it happens,” in *Proceedings of the 2005 ACM symposium on Software visualization*. ACM Press, 2005, pp. 115–124. [Online]. Available: <https://doi.org/10.1145/1056018.1056034>
- [42] F. Duseau, B. Dufour, and H. Sahraoui, “Vasco: A visual approach to explore object churn in framework-intensive applications,” in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, Sep 2012, pp. 15–24. [Online]. Available: <http://ieeexplore.ieee.org/document/6405248/>
- [43] D. A. Keim, “Information visualization and visual data mining,” *IEEE transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 1–8, 2002. [Online]. Available: <http://ieeexplore.ieee.org/document/981847/>
- [44] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich, “Abstracting runtime heaps for program understanding,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 774–786, Jun 2013. [Online]. Available: <http://ieeexplore.ieee.org/document/6331492/>
- [45] R. Ishizue, K. Sakamoto, H. Washizaki, and Y. Fukazawa, “Pvc.js: visualizing c programs on web browsers for novices,” *Heliyon*, vol. 6, no. 4, p. e03806, Apr 2020. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S2405844020306514>
- [46] “Guppy 3 home page,” accessed: 2022-01-27. [Online]. Available: <https://github.com/zhuyifei1999/guppy3>
- [47] “Identifying memory leaks - pypmpler documentation,” accessed: 2022-01-27. [Online]. Available: <https://pythonhosted.org/Pypmpler/muppy.html>
- [48] “Python object graphs,” accessed: 2022-01-27. [Online]. Available: <https://mg.pov.lt/objgraph/>
- [49] “Memory-profiler,” accessed: 2022-01-27. [Online]. Available: <https://pypi.org/project/memory-profiler/>
- [50] E. D. Berger, “Scalene: Scripting-language aware profiling for python,” 2020. [Online]. Available: <https://arxiv.org/abs/2006.03879>
- [51] “memray, a memory profiler for python,” accessed: 2022-01-27. [Online]. Available: <https://github.com/bloomberg/memray>
- [52] J. Kubelka, R. Robbes, and A. Bergel, “The road to live programming: insights from the practice,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1090–1101.
- [53] T. Fritz and G. C. Murphy, “Using information fragments to answer the questions developers ask,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. ACM Press, 2010, pp. 175–184. [Online]. Available: <https://doi.org/10.1145/1806799.1806828>
- [54] B. De Alwis and G. C. Murphy, “Answering conceptual queries with ferret,” in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 21–30.
- [55] A. J. Ko, R. DeLine, and G. Venolia, “Information needs in collocated software development teams,” in *29th International Conference on Software Engineering (ICSE’07)*. IEEE, May 2007, pp. 344–353. [Online]. Available: <https://ieeexplore.ieee.org/document/4222596/>
- [56] T. D. LaToza and B. A. Myers, “Developers ask reachability questions,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM Press, 2010, pp. 185–194. [Online]. Available: <https://doi.org/10.1145/1806799.1806829>
- [57] A. Holzinger, “Usability engineering methods for software developers,” *Communications of the ACM*, vol. 48, no. 1, pp. 71–74, Jan 2005. [Online]. Available: <https://doi.org/10.1145/1039539.1039541>
- [58] G. Terry, N. Hayfield, V. Clarke, and V. Braun, “Thematic analysis,” *The Sage handbook of qualitative research in psychology*, pp. 17–37, 2017.
- [59] A. J. Viera and J. M. Garrett, “Understanding interobserver agreement: the kappa statistic,” *Fam. med.*, vol. 37, no. 5, pp. 360–363, May 2005.
- [60] B. Shneiderman, “The eyes have it: A task by data type taxonomy for information visualizations,” in *The craft of information visualization*, ser. Interactive Technologies. Elsevier, 2003, pp. 364–371.
- [61] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, “Effectiveness of end-user debugging software features: Are there gender issues?” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 869–878. [Online]. Available: <https://doi.org/10.1145/1054972.1055094>
- [62] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrence, A. Blackwell, and C. Cook, “Tinkering and gender in end-user programmers’ debugging,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 231–240. [Online]. Available: <https://doi.org/10.1145/1124772.1124808>
- [63] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.