

Building a Debugger for WebTigerPython

Stijve Theo

Bachelor's Thesis

2024

Supervisors: Prof. Dr. Dennis Komm
Alexandra Maximova
Clemens Bachmann

Abstract

This thesis presents the design and implementation of a debugger for WebTigerPython, a web-based Python programming environment developed by the Center for Informatics Education at ETH Zürich. Aimed at enhancing the learning experience for programming students, the debugger addresses a significant gap in WebTigerPython, which, despite its potential as an educational tool, lacked debugging capabilities essential for effective learning and code troubleshooting.

Building upon Python’s `bdb` and `dis` modules, the debugger provides key functionalities such as step-by-step code execution, breakpoint handling, and comprehensive visualization of the program’s memory state, including mutable variables, references and their interrelationships. The design emphasizes simplicity and user-friendliness to accommodate beginners, drawing inspiration from similar educational tools like WebTigerJython, Python Tutor and Thonny.

The implementation involves modifying program execution to allow users to pause and inspect code at critical points, facilitating a deeper understanding of program behavior. Memory state capture is achieved using introspection tools, and the visualization interface leverages web technologies like `Vue.js` and `LeaderLineNew` to present data in an accessible manner.

Evaluation through performance testing indicates that the debugger introduces a significant though acceptable overhead for typical educational use cases. A user study further demonstrates that the debugger effectively aids students in understanding and correcting coding errors for which they already have the conceptual basis, and provides a helpful visual support for teachers otherwise, thereby improving learning outcomes. Participants reported high usability and found the tool intuitive for exploring code execution and memory state.

The contributions of this work extend beyond the functional debugger to include insights into the challenges of integrating debugging tools into educational programming environments. Future work involves enhancing the debugger’s capabilities, such as improved support for complex data structures and additional visualization features, to further enrich the learning experience in WebTigerPython.

Acknowledgment

I would like to thank everyone who made this thesis possible.

In particular, I would like to thank Prof. Dr. Dennis Komm for the opportunity to write my thesis in this group, my wonderful supervisors who guided and supported me every step of the way, Cédric Donner for his valuable testing, feedback and insights, Tobias Antensteiner for sharing precious resources and pointing me in the right direction, as well as all of those who dedicated their time to completing the user study.

Contents

1	Introduction	1
1.1	WebTigerPython	1
1.2	Motivation	1
1.3	Main Contribution	2
2	Foundations	3
2.1	What is a debugger?	3
	Definition	3
	Forms and Components of a Debugger	3
2.2	Related Works	5
	WebTigerJython	5
	Tiger Jython IDE	6
	Python Tutor	7
	Thonny	7
	Conclusions	8
2.3	Design Choices	9
3	Building the Debugger	11
3.1	Modifying Program Execution	11
	Conceptual Outline	12
	Bdb	12
	Implementation	14
3.2	Memory State Capture	15
	Conceptual Outline	15
	The Inspect Module	16
	Implementation	17
3.3	Memory Visualization	18
	Conceptual Outline	18
	Framework and Tools	21
	Implementation	21
4	Evaluation	25
4.1	Performance Evaluation	25
	Recursion	25
	Matrix Multiplication	26
	I/O-bound Test	26
	Memory Management Operations	27

4.2	Usability Testing	28
1.	Introduction	28
2.	Demonstration of the Debugger Functionality	28
3.	First Questionnaire	28
4.	Explanation of the task	29
5.	Guess Exercises	29
6.	Free Exercise(s)	32
7.	Second Questionnaire	32
8.	Informal Discussion	32
4.3	Comparative Evaluation	32
5	Results	35
5.1	Functionality	35
	Current Functionality	35
	Current bugs/errors	36
	Current limitations	36
5.2	Performance	38
5.3	User Study	38
	Guess Exercises	38
	Free Exercises	40
	System Usability Scale	40
5.4	Methodological shortcomings	41
6	Conclusion	43
6.1	Key Findings	43
6.2	Future Work	44

Chapter 1

Introduction

This work seeks to explain how to build a debugger for WebTigerPython. However, before tackling the technical details of its implementation, we first need to explore the context in which it emerges and the needs it seeks to meet.

1.1 WebTigerPython

WebTigerPython is a web-based Python programming environment developed by the Center for Informatics Education at ETH Zurich [1]. It is the successor of WebTigerJython and is especially designed for students [2].

Like its predecessor, WebTigerPython is designed to support a wide range of programming activities, including basic syntax learning, algorithm development, and the creation of interactive applications like robotics control and data visualizations. It emphasizes a user-friendly interface tailored for beginners and computer science classrooms.

Although the primary goal of WebTigerPython is not merely to facilitate the integration of newer versions of Python but to surpass its predecessor, at the time of writing, it remains in a beta phase with several key features still pending transfer, including, most notably for our purposes, the debugger.

1.2 Motivation

Troubleshooting tools seem to be an integral part of many Python learning environments. WebTigerJython included the possibility for students to execute their code line by line with a comprehensive overview of the memory state. TigerJython [3] uses user-friendly error messages and includes data visualization utilities. Python Tutor [4] enables various methods of code visualization and Thonny Integrated Development Environment [5] lets, among other features, learners execute their code expression by expression and highlights syntax errors.

The ability to accurately identify and correct one's mistakes is a critical component of any learning journey. In the realm of programming, debuggers can play a pivotal

role in this process by offering immediate visual feedback on code execution, the state of variables, and the occurrence of errors.

Introducing a debugger into WebTigerPython seeks to equip students with intuitive and comprehensive tools to enhance their learning experience by making it both more productive and more engaging.

Even beyond learning environments, debuggers are crucial. It is estimated, varying between studies, that software developers spend up to 50% of their time "fixing bugs" or "making their code work" [6]. Cutting this time down, beyond the conceivable improvement of the programming experience, has obvious and important economical ramifications.

Undoubtedly, as in many field, detecting errors is of utmost importance in computer science because even small mistakes in code or logic can lead to significant failures, including software crashes, data corruption, security vulnerabilities, or unexpected behavior. In mission-critical systems like aerospace, healthcare, nuclear energy, and financial services, undetected errors can cause catastrophic consequences, including loss of life, data breaches, or financial collapse [7].

1.3 Main Contribution

The primary contribution of this work is the design and implementation of an operational model for an interactive debugger for WebTigerPython, which enhances the platform's capability to provide detailed feedback on user's code and aims at improving the overall learning experience for programming students.

I have been working on implementing in WebTigerPython and improving on WebTigerPython's debugging functionalities such as, in priority, enabling the user to execute the code line by line and comprehensively displaying all variables currently in memory with their respective values.

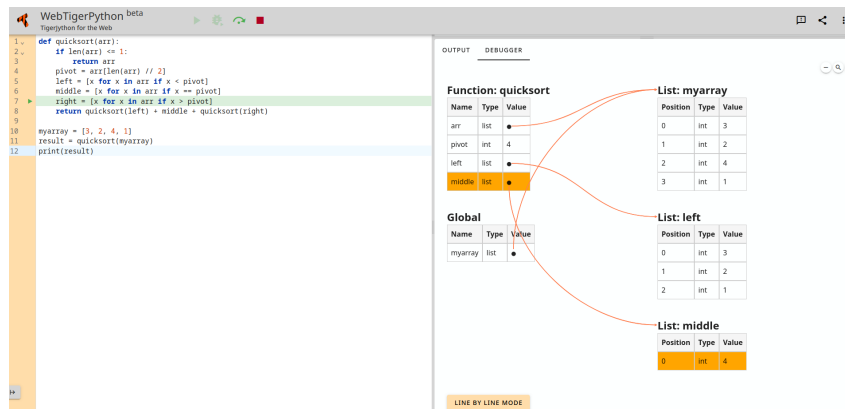


Figure 1.1. WebTigerPython's debugger during the execution of the quicksort algorithm.

Chapter 2

Foundations

This section strives to lay the framework in which to understand the key concepts, tools, and methodologies that underpin the development and implementation of the debugger.

2.1 What is a debugger?

As argued above, debugging plays an essential role in software development. One can define debugging as the process "consisting in localizing, understanding, and fixing software bugs, with the goal of making software systems behave as expected", [8]. Bugs encompass all kind of errors (logical, runtime, syntax, etc.) that result in the program either crashing or not executing as intended, sometimes with dramatic consequences [9].

If debugging is a process, a debugger can be understood as a category of tools for that process.

Definition

In this work, we understand a debugger as a program or collection of programs used to test, sometimes automatically correct, and most often better understand a program's behavior to ensure its proper functionality.

"Debuggers are the magnifying glass, the microscope, the logic analyzer, the profiler, and the browser with which a program can be examined."

— Jonathan B. Rosenberg, *How Debuggers Work* [10]

More technically, a debugger usually modifies the execution environment of a program to provide more information about the program, its environment and additional ways to interact and test it in order to facilitate the detection and resolution of bugs.

Forms and Components of a Debugger

Naturally, there exist multiple approaches, ways and utilities debuggers can include. Typical instruments include:

1. Syntax error highlighting.
2. Improved error messages (for example in order to make them more descriptive or explicit about what caused the error).
3. The ability to run/halt a target program, be it to execute it line by line, expression by expression, function call by function call, sometimes even backwards.
4. The ability to set breakpoints, sometimes conditional.
5. The display of the content in memory (variables, datastructures, stack, heap, etc.).
6. The ability to modify the content of the memory.
7. Exception handling.
8. Performance profiling.
9. ...

The following tools are less common but nonetheless sometimes integral parts of debuggers:

1. Control flow visualization.
2. Data relationship visualization.
3. Warnings for code smells (surface-level indicators of underlying issues in a software program's design or structure that, while not necessarily causing immediate errors, suggest the need for refactoring to improve maintainability and readability).
4. Memory leak, deadlock and race condition detection.
5. ...

The lists above are, of course, very far from being exhaustive and are largely inspired from Rosenberg's work [10]. Rather, they more humbly attempt to support the point that debuggers come in many forms and shapes depending on their context of execution and target audience.

It is worth noting that, generally speaking, there exists "little knowledge on how software engineers debug software problems in the real world" and that, generally speaking, among programmers, "both knowledge and use of advanced debugging features are low" [11].

Given the impracticality of implementing all possible tools, and the consequent need to prioritize which tools to include or exclude in WebTigerPython, let us inform this decision by examining how similar projects have addressed this challenge.

2.2 Related Works

In this section, we will explore the debugging tools and utilities implemented by comparable projects, that is Python IDEs aimed primarily at beginners, in their respective platforms.

WebTigerJython

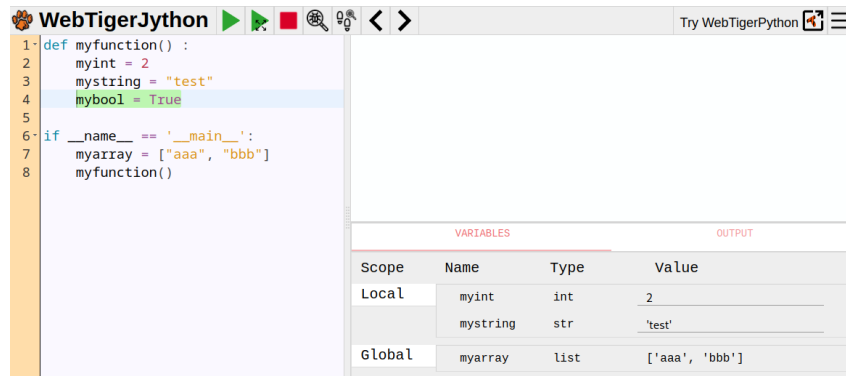


Figure 2.1. WebTigerJython's debugging utilities.

Our first source of inspiration of course will be WebTigerPython's ancestor project, WebTigerJython [12]. As shown in Figure 2.1, it includes two main features, namely the ability to execute the code one line at a time (back and forth) as well as a display of the variables in memory over two different scopes, "Global" and "Local". Error messages are made more accessible and, upon syntax errors, the line where the error took place is highlighted with a suitable suggestion for replacement. It is worth noting that statements which trivially have no effect on the program execution will be marked.

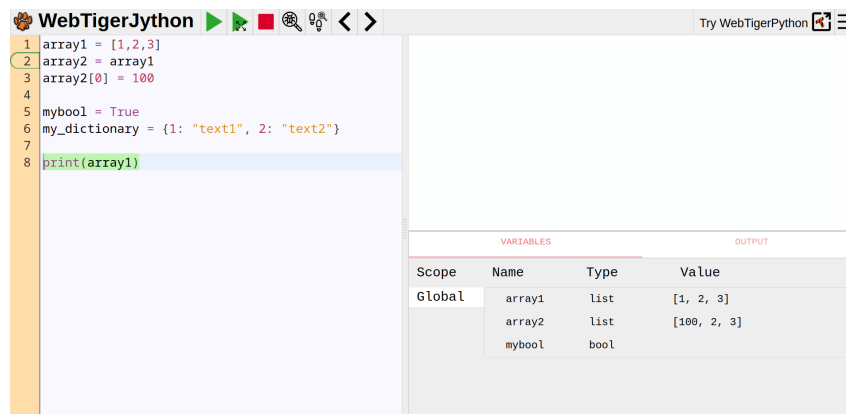


Figure 2.2. WebTigerJython's limitations.

This debugger is very simple which makes its usage intuitive. It has, however, important shortcomings. As displayed in Figure 2.2, it is unable to deal with some core data types, such as Booleans and dictionaries. Moreover, the debugger does not correctly display shared references to the same object. As we can see in Figure 2.2,

both `array1` and `array2` point to the same place in memory, but the debugger attributes them two different values. It is worth noting that since WebTigerJython enables backward execution, the debugger is not able to deal with user inputs within the program.

Tiger Jython IDE

Tiger Jython [3] in its desktop version offers a similar though somewhat more refined version of the same utilities.

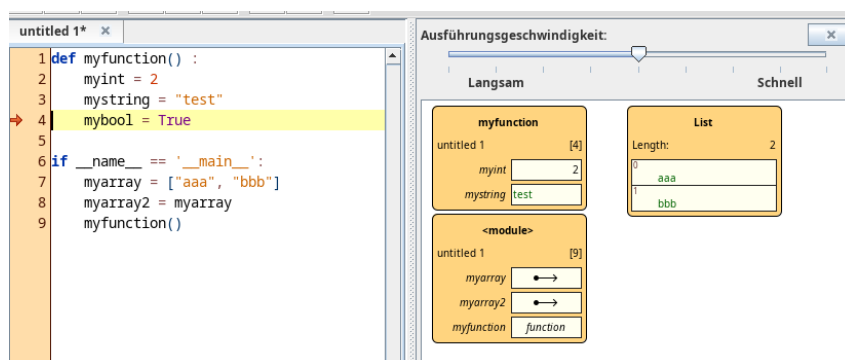


Figure 2.3. Tiger Jython’s debugger.

As depicted in Figure 2.3, a modular display enables a comprehensive view of all execution contexts. Line by line execution is also present. It possesses the same syntax highlighting and correcting assistance as WebTigerJython. Furthermore, it offers the possibility to load the last frame in the debugger upon error, as well as various statistics about the program execution.

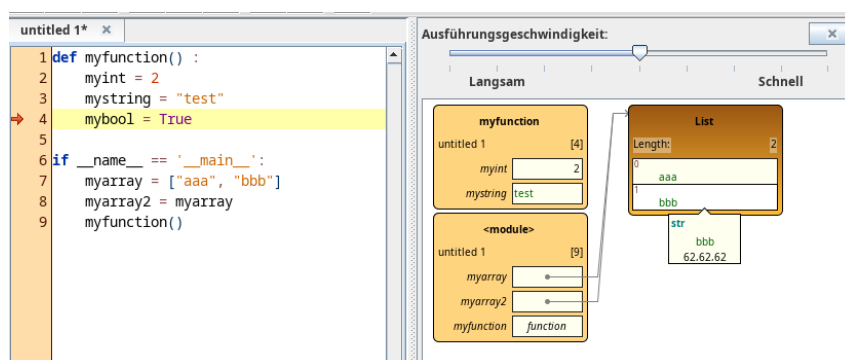


Figure 2.4. In Tiger Jython, additional information reveal upon hovering

When it comes to the display, hovering over elements will reveal their relationships through arrows, along with additional details like their type and memory location. One should note that it does not share the shortcomings of its web counterpart.

Python Tutor

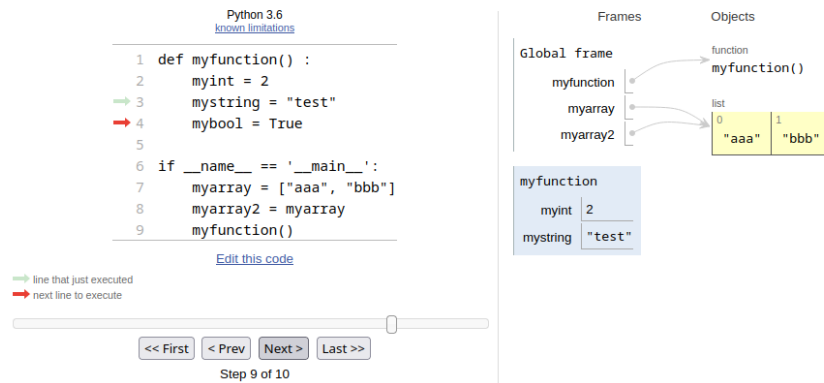


Figure 2.5. Python Tutor's debugger

Python Tutor [4] is somewhat of a special case in that it's a debugger before being a code editor. Once again, we can notice the ability to execute the code step by step, forward and backwards, as well as a comprehensive display of the content of the memory.

In order to make the display more readable, while in Python, all objects are, by their own admission, conceptually on the heap [13], they differentiate between frames and objects. References pointing to the same objects are displayed with arrows. Interestingly, functions are also treated as objects. The very modular and flexible nature of Python Tutor's display enables a very detailed and intuitive way of displaying complex and nested datastructures.

Python Tutor, however, has a number of limitations, the main one being that it cannot offer visualisation of long code.

Thonny

At last, let us examine Thonny [5], a Python IDE particularly interesting for our purpose as it is intended for beginners.

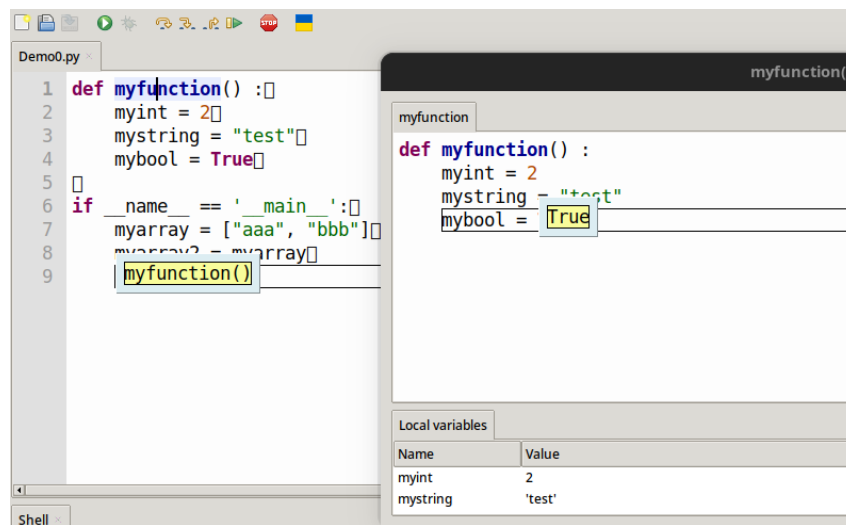


Figure 2.6. Thonny's debugging utilities

In Figure 2.6, we can notice that, like all previous examples, Thonny enables step by step execution. In this case, however, there is more to it than executing the code line by line. Thonny also offers the possibility to execute the code expression by expression. It moreover enables users to either:

1. Step over: This command runs the current line of code but skips over any function calls within that line. If the line contains a function call, it will execute the entire function and move to the next line in the current scope, without stepping into the function itself.
2. Step into: This command allows to step into a function or method call on the current line. Instead of executing the entire function in one go, the debugger will move into the function and pause at the first line inside it, allowing you to step through the function's execution line by line.
3. Step out: This command is used to step out of the current function. If one is currently inside a function, using "Step Out" will run the remainder of the function and bring back to the calling context (the next line after the function call in the original scope). It essentially exits from the current function and return to the caller.

When it comes to the visualization, it is interesting to note that Thonny makes the choice of opening every new called function in a new window, which offers an arguably intuitive way to understand the program's behavior in case of recursive calls. Every such window displays associated local variables by means of a table.

Conclusions

The analysis of debugging tools across platforms such as WebTigerJython, Tiger Jython, Python Tutor, and Thonny reveals several key aspects that can guide the development of WebTigerPython. One consistent theme is the balance between simplicity and functionality, where most debuggers stick to simple ways of modifying the execution flow as well as comprehensive displays of the content of the memory.

The most consistent functionalities appear to be the ability to execute the code step by step, where a step usually corresponds to a line of code, though not always. Additionally, the content of the memory is displayed accounting for the specific execution frame and shared references, often using modules, tables, and arrows to emphasize their relationships.

2.3 Design Choices

Considering that WebTigerPython is both running in the browser and a learning environment, it is crucial that the implementation of a debugger remains reasonably lightweight as well as its output perfectly readable, even to the untrained eye. Revisiting the definition explored in [Section 2.1](#), the implementation of debugging utilities in WebTigerPython will naturally particularly focus on helping students improve their understanding of the program's behavior and let other aspects such as enabling extensive testing, extended automatic error detection and conceptual visualization take the back seat.

More concretely, when building the debugger, we will aim at providing the following functionalities:

1. Syntax error highlighting. (Already existing)
2. Improved error messages (for example in order to make them more descriptive or explicit about what caused the error). (Already existing)
3. The ability to run/halt the target program, primarily line by line as WebTigerPython provided but also, in order to make it usable in case of longer programs as well, breakpoint by breakpoint
4. The display of a selected content of the memory, namely the variables, datastructures and classes that the user defined.
5. The display of relationships between and the execution context of this selected content of the memory.

Chapter 3

Building the Debugger

In this chapter, we dive into the core contributions of this work: the design and implementation of a debugger for WebTigerPython.

At the time of the start of this project, WebTigerPython already came with friendly error messages and error tracing. It will thus focus on modifying the program execution to enable more user control, capturing a selected sample of the memory and rendering it comprehensibly.

In order to change the user's code original execution behavior and allow for deeper inspection, the debugger is built on top of the user code that is literally passed around as a variable to it, as shown by the code snippet [Listing 3.1](#)

3.1 Modifying Program Execution

Modifying program execution is the most fundamental feature of our debugger, as it allows us to pause, inspect, and control the flow of a program as it runs. In WebTigerPython, we achieve this through the development of the **StepDebugger** class, which builds upon Python's `bdb` module [14] which provides basic hooks into the execution of a Python program, allowing us to halt at certain lines, inspect variables, and control the flow of execution.

```
1 async def start_debugger(code):
2     safe_code = repr(code)
3
4     code_to_debug = f"""
5 from debugger import execute_code
6
7 init_code = {safe_code}
8
9 execute_code(init_code)
10 """
11
12     return code_to_debug
```

Listing 3.1. Code snippet for starting a debugger with safe code embedding.

Conceptual Outline

The foundation of this debugger lies in modifying how Python code is executed in the WebTigerPython environment. By building on top of Python's `bdb` (Basic Debugger) module, we are able to intercept and control code execution in a way that is intuitive for students, while maintaining flexibility for future extensions. A key contribution here is the ability to execute the code in different modes: line-by-line and breakpoint modes, giving users control over how they navigate through their code.

The primary objective of modifying program execution is to give the user fine-grained control over how and when the program runs. This involves breaking down code into smaller steps and allowing users to move through these steps at their own pace, fostering understanding. When for example working with Turtle Graphics [15], one of WebTigerPython's fundamental pedagogical features [16], students can witness how each line of code modifies their drawing and in which ways. Combined with the memory state capture discussed in Section 3.2 and the visualization discussed in Section 3.3, it aims to allow users to examine their code in details and discover what it actually does.

Bdb

Python's `bdb` module forms the basis for many debuggers, including the well-known `pdb`, Python debugger [17]. It provides hooks into the Python interpreter and provides a low-level interface for controlling the execution of Python programs, allowing developers to step through code execution, set breakpoints, and examine the state of the program at various stages. More importantly for the purpose of this work, it facilitates the creation of custom tools for monitoring and controlling program execution.

The core of `bdb` revolves around the `Bdb` class, which implements methods for tracing and controlling the execution flow of Python programs. Below are the key components and functionalities of the `bdb` module:

Trace Function: At its heart, the `bdb` module sets a trace function that monitors program execution. This function is called on every event, including when a line of code is executed or when a function is called or returned. The `Bdb` class uses this trace function to determine whether to pause execution, continue running, or step into a function.

Breakpoints: The `Bdb` class allows users to set breakpoints at specific lines in the code. A breakpoint tells the debugger to pause execution when the program reaches a certain point. Breakpoints can be conditional or unconditional. Conditional breakpoints pause execution only when a certain condition evaluates to `True`. Breakpoints are set using the `set_break()` method, which specifies the filename and line number where execution should stop. The `check_breakpoints()` method checks whether the current line corresponds to a breakpoint, and if so, the program execution is paused.

Stepping and Flow Control: The `bdb` module provides several methods to control the flow of the program during debugging:

- `set_step()`: This method allows the program to execute one line at a time, effectively stepping through the code.
- `set_next()`: This continues execution until the next line in the current function.
- `set_return()`: This allows the program to continue until the current function returns, effectively stepping out of the function.
- `set_continue()`: This resumes execution until the next breakpoint or the end of the program.

Call Stack Management: The `Bdb` class manages the call stack during program execution. It keeps track of the functions being called and allows users to inspect the stack frames, helping to analyze the flow of the program. The `bdb` module provides methods to inspect the current call stack and to display information about the active function and local variables.

Handling Exceptions: The trace function in `bdb` can also be used to detect and handle exceptions. When an exception is raised, the debugger can catch it and pause execution, allowing the developer to inspect the state of the program at the moment of failure.

The `bdb` module in Python is fundamentally built upon the `sys.settrace()` function, which allows the setting of a global trace function that gets called on various events during the execution of a program. However, `sys.settrace()` introduces a significant performance overhead as it is called on every single event that occurs during the execution of a program, such as the execution of a line of code, a function call, function return, and exceptions. This frequent invocation of the trace function means that, for every step in the program, the Python interpreter must halt and transfer control to the tracing function, which then processes the event. As demonstrated by the [Figure 3.1](#), `sys.settrace()` is fairly complex and this process disrupts the natural flow of the program, leading to increased execution time [18]. Furthermore, since Python is an interpreted language, the tracing mechanism adds additional layers of function calls, which further slow down execution. In the context of halting the execution flow in controlled ways being the goal, this performance overhead is not a major problem. However, when growing the project further, we might want to use another less intrusive technology to inspect the content of the memory, a feature that might be useful in different contexts than halted execution, that is why the memory state capture as in [Section 3.2](#) is built on another technology entirely.

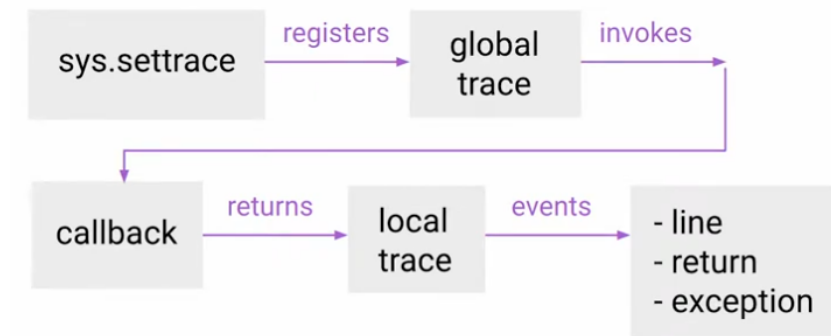


Figure 3.1. An overview of the functioning of `sys.settrace`.

Implementation

Modifying the execution flow of the program is accomplished by creating an instance of the `StepDebugger` class, which overrides specific methods within `bdb` to handle line execution, function calls, and exceptions in a user-friendly manner.

The `StepDebugger` class is a custom debugger that extends Python's `bdb.Bdb` class, providing enhanced functionality for debugging Python code, specifically controlling how code execution is traced line-by-line, with options for breakpoint-based control. Below is a breakdown of how it works:

Initialization (`__init__`): When a `StepDebugger` instance is created, it initializes the parent `bdb.Bdb` class and stores the code to be debugged as a list of lines using `self.lines = code.splitlines()`. This allows faster line retrieval than fetching lines from the execution frame itself. The `skip_next_line` flag is used to for example to avoid executing specific lines twice, particularly during class initialization, which Python handles in two steps. The `mode` parameter determines the behavior of the debugger, either stepping through code line-by-line or only stopping at lines marked with a breakpoint (i.e., containing `"#bp"` or `"# bp"`). The mode can be changed whenever the program is paused.

Line Execution (`user_line`): This method is invoked every time the debugger reaches a new line of code. The line number is extracted, and if the line falls within the valid range of the code, it checks the `skip_next_line` flag to avoid redundant execution. In breakpoint mode, the debugger only pauses at lines marked with `"#bp"` or `"# bp"`. If no such marker exists, the debugger continues to the next line using `self.set_next(frame)`. The method provides detailed information about the current frame's local variables (as discussed in [Section 3.2](#)), which is then passed to a callback function (`defaultrunner.callback`) in order to make them available to the front-end. Finally, `self.set_next(frame)` ensures the debugger continues execution without stepping into other functions or details like I/O operations, unless explicitly required.

Function Return Handling (`user_return`): This method is triggered when a function returns. It inspects the frame and captures the return value, printing relevant

details and passing the information to the callback. In breakpoint mode, it skips additional stops at return points and continues execution with `self.set_next(frame)`.

Exception Handling (`user_exception`): When an exception occurs, this method captures details about the exception type and message, prints them, and relays the information through the callback. The debugger continues execution after handling the exception.

Function Call Handling (`dispatch_call` and `user_call`): The `dispatch_call` method handles function calls. It ensures that only user-defined functions (within the `__main__` context) are traced. The `user_call` method is then invoked when entering a function, handing control over to `user_line` to step through the function's code.

Overall, the `StepDebugger` provides the basic structure for our debugger. The main challenge of the current implementation is to hide a part of Python's complexity to the user. Indeed, the user's code is not the only thing that gets executed when pressing the run button. Imported modules or complex functions have their own ramifications. At the moment, in order to verify whether a function is user-defined, and thus should be stepped into rather than over, we use the following condition in the `dispatch_call`: `if frame.f_globals.get('__name__') == '__main__':` (that is, the condition checks if the current script is being executed as the main program (i.e., not imported as a module), by verifying whether the global variable `__name__` is equal to `__main__`). This solution is less than ideal and might not scale well when WebTigerPython will adopt multi-file capabilities and let users import their own functions.

3.2 Memory State Capture

We here understand Memory State Capture as the process of recording the current state of variables, objects, and data structures at a particular moment during program execution. Building on the modified execution flow of the program discussed in [Section 3.1](#), for us, it'll mostly mean whenever the program is halted, either after the execution of a line of code or when a breakpoint has been reached. Conceptually, however, it is a completely separated process that could also happen independently and be useful in a wide range of different contexts.

Conceptual Outline

The core idea behind the implementation of memory state capture in this work is to gather a consistent and structured representation of variables and their types within the program's current frame. The goal is to focus only on variables that are actively involved in the current execution and present them in a way that is both human-readable and informative. This process includes creating representations for complex data structures such as lists, dictionaries, and objects, making it easier to understand the memory state in a dynamic program.

In more than one way, the content of the memory contains too much information

for the average Python learner to make sense of. Python frames are populated with objects that we would rather hide. Global variables defined in modules and packages can clutter the view, function context variables, closures and internals should probably remain internal, built-in and imported types and functions might add more confusion than help the understanding of the code. While ultimately all this information could turn out incredibly informative, the choice was made, given the pedagogical nature of the project, to only display user-defined variables, classes and methods, and to hide the rest from their view.

The Inspect Module

The `inspect` module in Python [19] provides a set of introspection tools that enable retrieving detailed information about live objects, including modules, classes, functions, methods, and code frames. At its core, the `inspect` module can access function signatures, get details about class structures, retrieve the source code of functions or methods, and provide information about the stack frames during execution.

One of the most important uses of the `inspect` module is its ability to examine the call stack using the `inspect.stack()` function. This function returns a list of `FrameInfo` objects that describe the current execution stack, including local and global variables, and the lines of code being executed.

When it comes to capturing memory, the `inspect` module has several key advantages over `bdb`, most notably in terms of performance. Indeed, Python's `inspect` module operates intrinsically close to the core of Python's interpreter, enabling it to perform introspection tasks with minimal overhead. In contrast, the `bdb` module introduces considerable overhead by relying on Python's `sys.settrace()` function to install trace hooks that are invoked on every significant event during the execution of a program [18]. Each of these trace hooks involves additional function calls and context switching, which collectively result in a substantial performance penalty, as discussed in [Section 3.1](#).

Since the `inspect` module's design emphasizes passive observation rather than active intervention in the program's execution it accesses existing execution frames and object states without modifying or controlling the flow of the program. Additionally, `inspect` benefits from Python's optimized handling of introspection tasks. Many of the functions within `inspect` are implemented in C, providing an efficient interface for accessing low-level details of Python objects and execution frames.

Additionally to the `inspect` module, we use the `dis` module (short for disassembly) to efficiently inspect Python's bytecode and filter for instructions that are of particular interest, i.e., instructions that load, store or delete data. The `dis.dis()` function disassembles Python functions, methods, or classes and outputs the corresponding bytecode instructions [20].

Implementation

The memory capture, and then its systematic filtering, happens in multiple steps.

Type Map In order for our display to be as consistent as possible, the program contains a hard-coded type map, mapping variable types to a unified representation. Indeed, in some contexts, Python might use different names for the same types. For example, while Python’s standard type for integers is `int`, in some contexts, such as older versions of certain libraries or interfaces, the type can be referred to as `integer`.

Safe representation It can happen that complex objects are not representable, that is, that Python cannot generate a meaningful or valid representation for those objects. That can for example be the case for circular or custom objects as well as pointers or threads or deeply recursive types. In order to avoid a crash, we first ensure that the objects we are dealing with are representable.

Propagation of mutable objects Mutable objects in Python are objects that can be modified. Examples of such objects are lists, dictionaries, sets. Mutable objects are usually, from the programmer’s point of view, best understood as a pointer to a place in memory. There can be multiple pointers to that same place in memory with different names. Where it, in our case, becomes delicate, is that mutable objects can be defined at a lower stack frame and be accessed at higher ones. As our method of tracking variables includes only looking at the current stack frame, it is possible that a mutable object is modified without this modification being visible until all functions return and we find ourselves back at the original stack frame where the mutable was defined. This might however be too late for the programmer to really be able to understand what changes said mutable object went through. For this reason, we artificially propagate the reference of mutable objects through all relevant stack frames.

print_local_variables The `print_local_variables` function is our main function, wrapping all of the utilities above. It receives a frame as an argument from the `StepDebugger` class defined as in [Section 3.1](#). It is however possible to get the same object through `inspect.currentframe()` or `inspect.stack()` in case one wants to run it in a different context. Then, it processes the object as follows:

```
1     local_vars = frame.f_locals
2     instructions = dis.get_instructions(frame.f_code)
3
4     active_vars = {instr.argval for instr in instructions if instr.
                    opname.startswith(('LOAD', 'STORE', 'DELETE'))}
```

- `local_vars = frame.f_locals`: This line retrieves the local variables of the current execution frame. The `frame.f_locals` dictionary has the names of local variables as key and their current value as associated values. This provides access to all variables defined within the current scope.

- `instructions = dis.get_instructions(frame.f_code)`: The instructions are gathered using the `dis.get_instructions()` function which disassembles the bytecode of the function or method associated with the current frame's code object, which is accessed through `frame.f_code`. This function returns an iterator over the bytecode instructions, allowing us to inspect each operation that Python will execute. These instructions contain details like the operation name (e.g., `LOAD`, `STORE`, or `DELETE`) and the argument values (which typically refer to variable names).
- `active_vars = The line {instr.argval for instr in instructions if instr.opname.startswith(('LOAD', 'STORE', 'DELETE'))}`: creates a set of active variables. It iterates over the disassembled instructions, specifically looking for operations that deal with loading, storing, or deleting variables. The bytecode operations `LOAD`, `STORE`, and `DELETE` correspond to loading a variable onto the stack, storing a value into a variable, or deleting a variable, respectively. For each relevant instruction, the argument value (`instr.argval`) is extracted, which corresponds to the variable name involved in the operation. These variable names are then stored in the `active_vars` set, representing the variables that are actively being manipulated in the code. Those are of particular interest since they might affect the content of user-defined variables and datastructures which we want to display.

Filtering The `print_local_variables` function then methodically filters out undesired variables such as built-in functions, methods, etc. to only keep user-defined ones and formats them in a consistent way depending on their type.

Return values The `print_local_variables` function returns the stack depth and function name in order to give the relevant context of execution to the user. Additionally and more importantly, it returns the filtered variables as a dictionary. The keys of the dictionary are the variable names and their associated value is a dictionary itself with the value, a safe, detailed representation of the variable, a type and an id which consists of the memory location of the variable. This id is particularly important to identify multiple variables sharing a common reference.

3.3 Memory Visualization

Once the relevant memory information is captured as described in [Section 3.2](#), it is sent to the front-end through a callback. This section discusses how this information is processed and then rendered to display the content of the memory with the primary goal of fostering a clear understanding of the program's behavior.

Conceptual Outline

In Python, when an argument is passed to a function, it is passed by **object reference**. This means that the function receives a reference to the object in memory, but the behavior differs based on whether the object is mutable or immutable. If the object is mutable (as for example lists or dictionaries are), the function can modify

the object in place and these changes will persist outside the function. However, if the object is immutable (as for example integers or strings are), attempts to modify them within the function will create a new object, leaving the original unchanged [21].

When displaying the content of the memory to the user, we of course want to make it visually obvious when two variable-names point to the same object in memory. Technically speaking however, in Python, even two integers with the same value are two references to the same object. A simple way to convince oneself of this is to run the Python code below.

```
1 a = 100
2 b = 100
3 print(a is b)
```

It will indeed always return True, and the same behavior can be observed for Booleans, floats, strings, etc. Interestingly, when it comes to tuples (which in short are non-mutable lists), not all Python environments seem to agree. Python Tutor for example will return False. This difference is of course reflected visually. While the debugger in WebTigerPython will point at the tuples as being the same object, as shown in Figure 3.2, Python Tutor will create two separate entities, as displayed in Figure 3.3. In their "known limitations" document, they suggest this behavior might be "implementation-specific and differ[ing] between REPL and scripts" [13].

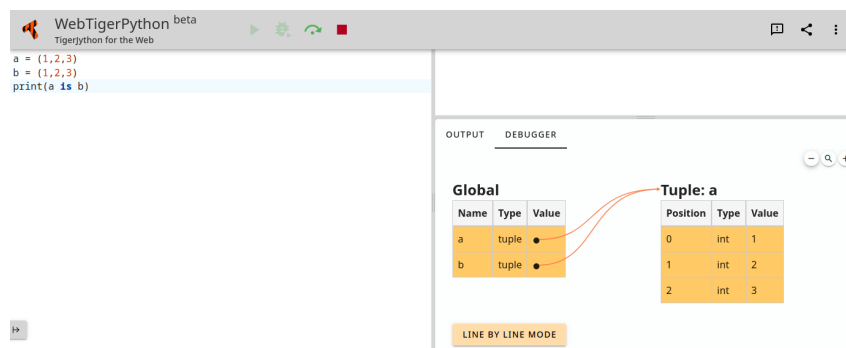


Figure 3.2. Referencing immutable objects in WebTigerPython

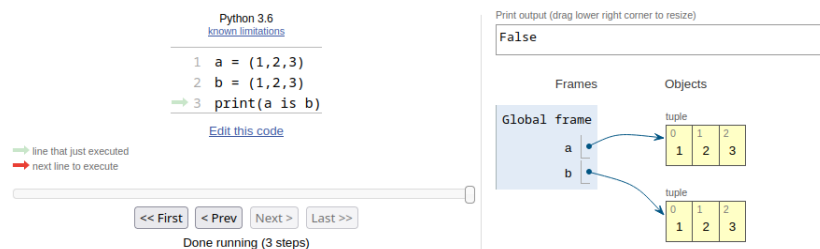


Figure 3.3. Referencing immutable objects in PythonTutor

For mutable objects however, creating two objects with the same values will in fact

create two different objects in memory. The code below will evaluate to false.

```

1 a = [1,2,3]
2 b = [1,2,3]
3 print(a is b)

```

It is worth noting however that the same mutable object in memory can of have an arbitrary number of variables referencing it, potentially modifying it across all stack frames. A write to any of those references propagates changes the value their all contain. On one hand, we would like to adequately represent the reference relationship between the objects in memory and on the other, it is crucial to keep the display as readable as possible, otherwise the debugger will be rendered useless.

In this project, arrows are used to indicate that a variable references an object. However, this notation is applied only to complex objects, meaning objects that can hold more than a single value. To avoid cluttering the display, simple data types—such as integers, booleans, strings, and floats—are represented as a line in a concise three-column table, displaying the variable name, type, and value whereas the lines related to lists, tuples, dictionaries, sets, frozensets, instances of user-defined classes, etc. instead of directly denoting the value in the value-column rather point to a separate associated table with the details of their content. Two references to the same object will point to the same associated table. Moreover, to outline the different execution contexts, a new table is created for each different currently active stack depth. This attempt to balance technical truth and legibility is exemplified by the [Figure 3.4](#).

For the moment, the choice has been made not to display detailed development and relationships between the content of such complex objects as arbitrary depth and vertical or "backward" (here to understand as right to left) arrows might, unless are given special treatment beyond the scope of this work, confuse rather than clarify.

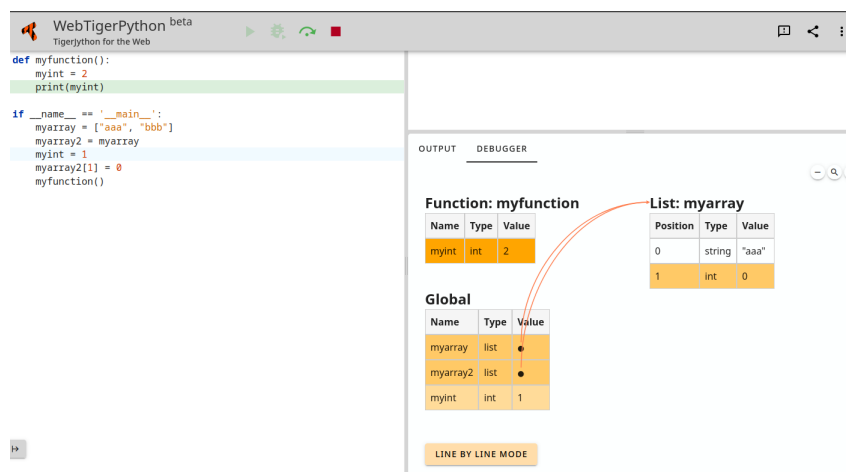


Figure 3.4. Rendering balance

Framework and Tools

WebTigerPython uses multiple technologies to create an interactive learning environment, including Vue.js for the front-end interface, Pyodide for running Python in the browser, Pinia for state management, and Vuetify for building a responsive and visually appealing user interface. For the specific need of the debugger, Lodash for utility functions and LeaderLine for visual connections between elements have been added.

1. Vue.js: Vue.js is a JavaScript framework used for building user interfaces [22]. When it comes to the implementation of the debugger, the code uses Vue 3's `Composition` API to manage application state and reactivity. In particular, `ref()` function from Vue is used to create reactive references for variables, allowing seamless updates to the interface when the underlying data changes. The `computed()` function is employed to derive reactive data from existing state, ensuring that any changes are reflected immediately in the UI without manual intervention.

2. Pinia: Pinia is a state management library designed for Vue.js, acting as a global store for managing application-wide state [23]. The debugger has its own store called `debuggerStore`, keeping holds of the current state of the stack, the variables that have been passed on and object references. This store is written in TypeScript.

3. Pyodide: Pyodide brings Python to the browser by compiling it to WebAssembly and is the main technology underlying WebTigerPython [24]. The debugger, however, does not directly interact with or modify the behavior of this technology.

4. Lodash: Lodash is a utility library that simplifies common JavaScript operations, particularly those involving arrays, objects, and functions [25]. In our code, the `_.isEqual()` function from Lodash is used to perform deep equality checks between objects or variables. This ensures that the application can detect changes in variables, even if they are complex data structures like objects, arrays, or dictionaries in an established and well-tested way.

5. Leader-line-new: `Leader-line-new` is a JavaScript library used for drawing lines between HTML elements. It is the successor of `LeaderLine` [26]. This is useful for visually linking related pieces of information. In our case, `LeaderLine` is used to draw lines between variables in the stack (on the left side of the interface) and their detailed representations (on the right side) as exemplified in [Figure 3.4](#). They for example link list names with their detailed representation. Since multiple list names can point to the same list, the arrows visually display this relationship with the goal of clarifying the role of references to the user.

Implementation

As discussed in the previous subsections, the memory visualization implementation of the WebTigerPython debugger is built to track and display object references across different stack frames while attempting to maintain clarity for the user. The

debugger distinguishes between two categories of variables: stack variables and detailed variables. Stack variables are local variables specific to each function or scope, captured as snapshots at different depth in the calls tack. Each stack frame includes a collection of variables associated with its depth. Detailed variables on the other hand are used to represent more involved data structures like lists, dictionaries, sets, tuples and user-defined classes. These are handled separately and depending on their type to show their internal structure in a clear and organized way.

stackVariables is the core data structure for managing stack variables. It holds an array of objects, each representing a particular depth in the call stack, containing variables and their corresponding data. When a function returns and the stack depth lowers, its content is deleted accordingly.

detailedVariables contains the entries for complex objects, ensuring that large or nested data types are not flattened into a single table but instead represented in their own tables.

highlightedVariables is responsible for the "fade effect" used to visually emphasize changes in variables. The variables that have recently been updated are highlighted more strongly with the highlighting gradually fading over time, disappearing after 4 new line of code, or breakpoint, depending on the mode, executions.

Tracking changes: To track changes between different execution steps, the debugger utilizes the `.isEqual()` function from Lodash to perform deep equality checks between the current and previous states of variables. This ensures that even subtle changes within complex objects are detected. Using a well-tested library function ensures that these checks don't cause any error even in case of complex or unexpected datastructures. When a change is detected, the **highlightedVariables** mapping is updated and the fade level for the affected variables is reset to the maximum value.

Drawing arrows: The linking of stack variables to their detailed representations is handled using the LeaderLine library. Each complex object is assigned a unique identifier based on its memory location (`id`), variable name and stack depth. Arrows are being drawn between the variable's representation in the stack (on the left side of the interface) and their associated detailed structure (on the right side of the interface). This is done using dynamic DOM elements that serve as anchors for these arrows. The current implementation ensures that arrows are only drawn when both the variable and the title of the detailed representation are visible in their respective scrollable containers. The arrows are updated each time the user interacts with zooming, srcolling or stepping through the code.

Mode Change and Abort: Changing mode between line by line execution and breakpoint mode can be done whenever the program is paused. The part of the debugger described in [Section 3.1](#) issues a callback to the front-end and waits for its resolution before moving on with the execution. We resolve it sending a message containing either `'breakpoint'`, `'line by line'` or `'abort'`. The first two options

occur when the user presses on the “next step” button depending on the mode that is currently enabled. Changing the mode happens through pressing on another button, which itself is not associated with any direct response. Aborting also resolves the callback and happens when the user pressed on the normal program interrupt, enabling a clean new start.

Additionally, the system employs a cleanup process to remove unreferenced details from memory. Each time the stack state is updated, the debugger checks which objects are still referenced by active variables. If an object is no longer referenced by any stack variable, it is removed from the **detailedVariables** array and the associated **objectReferences** map ensuring efficient memory usage and preventing the display from becoming cluttered with obsolete data.

Chapter 4

Evaluation

The following chapter aims to evaluate three major aspects of the debugger. As a first step, the impact of the debugger on the performance will be discussed. Secondly, a methodology for a user study will be described before finally giving way to a comparison with its predecessor, WebTigerJython's debugger.

4.1 Performance Evaluation

In this section will be discussed the difference in performance, measured in execution time on the same machine, namely a Lenovo ThinkPad T480s laptop with an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz processor, featuring 4 cores and 8 threads, on an x86-64 architecture, and in the same browser, Firefox for Manjaro Linux 130.0 (64-bit). It was tested locally and no other resource-intensive process were running at the same time. The code snippets below have been executed five times with the debugger disabled and five times with it enabled, in breakpoint mode, without any breakpoints set (thus, without any stops).

Recursion

```
1 import time
2
3 def fibonacci(n):
4     if n <= 1:
5         return n
6     else:
7         return fibonacci(n-1) + fibonacci(n-2)
8
9 if __name__ == "__main__":
10     start_time = time.time()
11
12     n = 35 # Value changed between 20, 30 and 35
13     result = fibonacci(n)
14     print(f"Fibonacci number for {n} is {result}")
15
16     end_time = time.time()
17     print(f"Execution time: {end_time - start_time} seconds")
```

Listing 4.1. Benchmark program that measures the runtime of a recursive function computing Fibonacci numbers.

The code snippet in [Listing 4.1](#) demonstrates the overhead coming with using `bdb` univocally. For $n = 20$, running it without the debugger takes between 0.002 and 0.004 seconds while running with it takes between 0.23 and 0.25 seconds. For $n = 30$, the execution takes between 0.48 and 0.52 seconds in standard mode while in debugger mode, it takes from 28 to 31 seconds. The trend does not improve for $n = 35$ where a debugger-less run takes approximately 5.15 to 5.25 seconds while, with it, it takes around 5 minutes (298 to 310 seconds).

While the simplistic measurements do not permit the precise determination of the factor by which the debugger slows the code down; it is safe to say that it lies between 60 and 100, which is consequent. It is worth noting that this overhead primarily comes from the high number of Python operations being executed. In the case above, we execute the same lines over and over, calling the heavy `sys.settrace()` function over and over. In a more standard benchmark using matrix multiplication as below, no statistically significant difference between running with the debugger enabled and disabled can be noticed.

Matrix Multiplication

```

1 import numpy as np
2 import time
3
4 def matrix_multiplication_test(size):
5     A = np.random.rand(size, size)
6     B = np.random.rand(size, size)
7
8     start_time = time.time()
9     result = np.dot(A, B)
10    end_time = time.time()
11
12    print(f"Matrix multiplication of {size}x{size} matrices took: {
13          end_time - start_time:.6f} seconds")
14
15 matrix_multiplication_test(10)
16 matrix_multiplication_test(100)
17 matrix_multiplication_test(1000)
18 matrix_multiplication_test(2000)

```

Listing 4.2. Benchmark program that measures matrix multiplication time for random matrices of different sizes.

The code snippet in [Listing 4.2](#) indeed run in around 0 seconds for 10x10 matrices, 0.002 seconds for 100x100 matrices, between 2.7 and 3.3 seconds for 1000x1000 matrices and between 42 and 49 seconds for 2000x2000 matrices. The debugger here does not significantly impact the performance. This can probably be explained as `np.dot()` is heavily optimized and the heavy lifting done in lower level languages, not involving a lot of purely Python operations [27].

I/O-bound Test

```

1 import time

```

```

2 import os
3
4 def io_test(file_size_mb):
5     filename = 'test_io_file.txt'
6     data = 'A' * (file_size_mb * 1024 * 1024) # Prepare data of
       specified size
7
8     # Write file
9     start_time = time.time()
10    with open(filename, 'w') as f:
11        f.write(data)
12    end_time = time.time()
13    print(f"Writing {file_size_mb}MB took: {end_time - start_time:.6f}
       seconds")
14
15    # Read file
16    start_time = time.time()
17    with open(filename, 'r') as f:
18        _ = f.read()
19    end_time = time.time()
20    print(f"Reading {file_size_mb}MB took: {end_time - start_time:.6f}
       seconds")
21
22    # Clean up
23    os.remove(filename)
24
25 io_test(500) # Testing with a 500MB file

```

Listing 4.3. Benchmark program that measures I/O time for a 500MB file.

In this case again, we do not measure any significant difference between executing with and without the debugger. It is probably safe to assume that the main factor determining the performance is waiting for the system to complete I/O operations and that it is not impacted by `sys.settrace()`. Worth noting is that running the code snippet above with the debugger mode enabled and either one or more breakpoints on or a line by line execution will freeze and eventually crash as transmitting the content of the huge `data` variable is too much for the current implementation to handle.

Memory Management Operations

```

1 import time
2
3 def list_operations_test(size):
4     start_time = time.time()
5
6     # Create a large list
7     large_list = [i for i in range(size)]
8     end_time = time.time()
9     print(f"Creating a list of {size} elements took: {end_time -
       start_time:.6f} seconds")
10
11    # List slicing operation
12    start_time = time.time()
13    _ = large_list[:size//2]

```

```

14     end_time = time.time()
15     print(f"Slicing half the list took: {end_time - start_time:.6f}
16         seconds")
17
18     # Appending to the list
19     start_time = time.time()
20     large_list.extend([i for i in range(size)])
21     end_time = time.time()
22     print(f"Appending {size} elements took: {end_time - start_time:.6f}
23         seconds")
24
25 list_operations_test(10**7) # List size of 10 million elements

```

Listing 4.4. Benchmark program that measures memory management operations time for the creation, slicing and appending elements to a list of size 10^7 .

This last example confirms the findings of the other tests. The list creation and appending elements to the list have a large number of Python operations, which run in 0.6-0.7 seconds without, 29-31 seconds with the debugger enabled respectively. Slicing the list on the other hand is an optimized C operation and takes 0.02 seconds to run in both cases [28].

4.2 Usability Testing

In order to evaluate how users interact with the debugger and to assess whether it aids them identifying and resolving common coding mistakes as well as to discover potential shortcomings of it and inform future work, I have used the following experimental protocol on a sample of 6 students and Python learners. All participants were selected from my friends and relatives who have knowledge of Python but have neither studied computer science nor worked in a related field. Their ages range from 16 to 42 years old.

1. Introduction

The user first receives an explanation of the overall project, its goals and scope. They are then informed in detail on how the evaluation works and any questions they might have about either are carefully answered.

2. Demonstration of the Debugger Functionality

In this phase, I have shortly demonstrated how to interact with the debugger and its main capabilities. In particular, I have shown participants how to execute code line by line, how to set up a breakpoint and how to inspect the content of the memory.

3. First Questionnaire

Participants are then invited to fill out a brief initial questionnaire. It is composed of 3 questions, one aimed to collect their name, one to understand their current level of education and one where they self-assess their level of expertise with Python by choosing between one of the options below:

1. Complete Beginner – I am new to Python and just starting to learn basic concepts like variables, data types, and simple loops.
2. Novice – I can write small Python scripts, use loops and conditionals, and perform basic tasks, but I still need help with more complex functions and concepts.
3. Familiar user – I am comfortable with core Python concepts like functions, lists, and dictionaries, and I can solve common problems, but I’m still learning more advanced topics like file handling or working with external libraries.
4. Confident user – I can write Python programs that solve more complex tasks, work with libraries like pandas or matplotlib, and understand object-oriented programming (OOP) at a basic level.
5. Expert - I am proficient with advanced Python features like decorators, generators, multithreading, and asynchronous programming, and I can optimize code for performance and scalability.

4. Explanation of the task

It is then explained to the study’s participants that they will be submitted to a series of puzzle exercises based on the example below:

```
1 x = 1
2 y = 2
3 z = x + y
4
5 print(x)
6 print(y)
7 print(z)
```

They first have one minute to guess what the result of the print statements in the end will be. If they don’t find the answer or feel like they need additional information in order to solve the exercise, they are allowed to use the debugger for two minutes on their own before we ultimately start the discussion. Given the framework of our project given in [Section 2.1](#) of a debugger as a tool to foster understanding of the code, we only move on to the next puzzle once the participant fully understands the correct answer and carefully keep track of whether they came up with this understanding on their own, with the debugger or with the debugger and additional explanations.

5. Guess Exercises

The participants then proceed through the following exercises one at a time. Each exercise aims at testing the user’s understanding and ability to diagnose and correct common coding errors. The exercises were submitted to the participants in randomized order.

Different Execution Scopes

```
1 def modify_vars(x, y):
2     x = x + 1
3     y = y * 2
4     return x, y
5
6 x = 5
7 y = 10
8 modify_vars(x, y)
9
10 print(x)
11 print(y)
```

The example above is built in such a way that the intuitive answer is 6, 20. This is however not correct. It is not because `x` and `y` in the global frame and `x` and `y` in the function `modify_vars` have the same name that they are the same object. Conceptually, the debugger should be able to help come to this understanding by displaying each stack frame separately.

Out of Range

```
1 numbers = [0, 1, 2, 3, 4]
2 for num in range(5):
3     if num % 2 == 0:
4         numbers.pop(num)
5
6 print(numbers)
```

The example above is built in such a way that the intuitive answer is [1, 3]. The code above however will not successfully run and throw an index out of range exception. Indeed, when calling in the last iteration of the loop `numbers.pop(4)`, the list will already have shrunk to 3 elements. The intention is that the debugger can help come to this understanding by enabling a detailed step-through the loop.

Mutable Default Argument

```
1 def append_to_list(item, my_list=[]):
2     my_list.append(item)
3     return my_list
4
5 list1 = append_to_list(1)
6 list2 = append_to_list(2)
7
8 print(list1)
9 print(list2)
```

The intuitive answer for the piece of code above is [1], [2]. This is however incorrect, as the `append_to_list` function is initiated with a mutable argument `my_list`. This means that the first time the function is called, `my_list` is initiated to []. Then, when running `list1 = append_to_list(1)`, 1 is added to `my_list` and `list1` will now point at `my_list`. When running `list2 = append_to_list(2)`, 2 is added to `my_list` and `list2` will now point at `my_list`. The correct answer is thus [1, 2], [1, 2]. The intention is that the debugger, through arrow representation, is able to help participants understand the referential nature of variables representing mutable objects.

Off by One

```
1 def count_even(numbers):
2     total = 0
3     for i in range(len(numbers) - 1):
4         if numbers[i] % 2 == 0:
5             total = total + 1
6     return total
7
8 numbers = [0, 1, 2, 3, 4]
9 total = count_even(numbers)
10
11 print(total)
```

Off by one errors are very common types of programming mistakes. In the code snippet above, the intuitive result is 3 while the correct answer is 2 as `for i in range(len(numbers) - 1):` will only run from index 0 to index 3, ignoring the final even number "4". The intention is that the debugger can help detecting these kind of mistakes by stepping through the loop and noticing the value of the index at each iteration.

Mutables and Immutables

```
1 str = "hello"
2 copy = str
3 copy += "!"
4
5 original_set = {1, 2, 3}
6 copied_set = original_set
7 copied_set.add(4)
8
9 print(str)
10 print(copy)
11 print(original_set)
12 print(copied_set)
```

The exercise above mainly serves to demonstrate the difference between mutable and immutable objects in Python. The correct answer is of course "hello", "hello!", {1, 2, 3, 4}, {1, 2, 3, 4}, and in the case the participant was to make a mistake, they should be able to understand the difference between the two kinds of objects using the memory state capture abilities of the debugger.

Unexpected Collisions

```
1 d = {0: 'int', 1: 'int', 2: 'int'}
2 d[3] = 'int'
3 d[True] = 'boolean'
4 d["str"] = 'string'
5
6 print(d)
```

The puzzle in the code snippet above can be understood as a naive attempt to map variables to the string representation of their respective types using a dictionary. There is however a catch, as the key 1 and the key True are in reality the same key. The correct answer is {0: 'int', 1: 'boolean', 2: 'int', 3: 'int', "str": 'string'}.

Once again, the idea is that by carefully stepping through the assignments and watching the state of memory using the debugger, the collision becomes apparent.

6. Free Exercise(s)

This phase of the study is optional and was not carried out with all participants. The idea is that rather than observing their interactions with the debugger in examples crafted for that specific purpose, they were encouraged to freely code while their dealings with the debugger were noted. The suggested exercise was an implementation of the Fibonacci sequence given the following base:

```
1 def fib(n):  
2     #TODO, implement a function that returns the nth number of the  
   fibonacci sequence  
3  
4 n = int(input("What is the base number of the sequence?"))  
5 result = fib(n)  
6 print(result)
```

This exercise was chosen given its simplicity and its propensity to require some amount of debugging in order to get it fully right. There was however a case where a participant imported their own coding project into the interface and debugged it there, enabling an even more faithful recollection of the debugger's usage.

7. Second Questionnaire

After completing the exercises, participants were invited to fill out a second questionnaire composed of a measure of the System Usability Scale [29] and three additional questions concerning their experience using the debugger.

8. Informal Discussion

The evaluation concludes with an informal discussion allowing participants to share their thoughts in a more conversational format. The intent of this phase is to reveal insights on concrete user frustrations, unexpected findings or suggestions for improvements. This discussion is methodically written down.

The structured experimental protocol above is designed to assess the debugger's impact on user learning, first without guidance, than with it, as well as to evaluate the usability of the debugger's interface and identify concrete improvement areas.

4.3 Comparative Evaluation

As discussed in [Section 2.2](#), WebTigerJython [12] enabled users to step through the code line by line, back and forth, as well as display the content of the memory in forms of two tables, one for the global scope and one for the local scope. While the debugger in WebTigerPython is not able to execute the code backwards, it is improving on its predecessor in the following ways:

1. WebTigerJython's debugger is not able to deal with the `input()` function. WebTigerPython's debugger handles it properly.
2. WebTigerJython's debugger is not displaying boolean variables whereas WebTigerPython does.
3. WebTigerJython's debugger does not handle references to mutable objects correctly, as displayed in [Figure 4.1](#). In WebTigerPython, these cases are handled properly and visualized using separate tables and arrows.
4. WebTigerJython's debugger only comprehends two scopes (local and global) while in WebTigerPython, the scopes of all functions that have not returned yet are included.
5. WebTigerPython has the additional option of letting users step through the code breakpoint by breakpoint.
6. WebTigerPython's display puts an emphasis on the last four items in memory that were changed using fading orange highlighting and enables users to zoom in and out of the debugger display.

Arguably, WebTigerPython's debugger display is more involved which has both the advantage of offering more information to users who know how to use it while being slightly less novice-friendly.

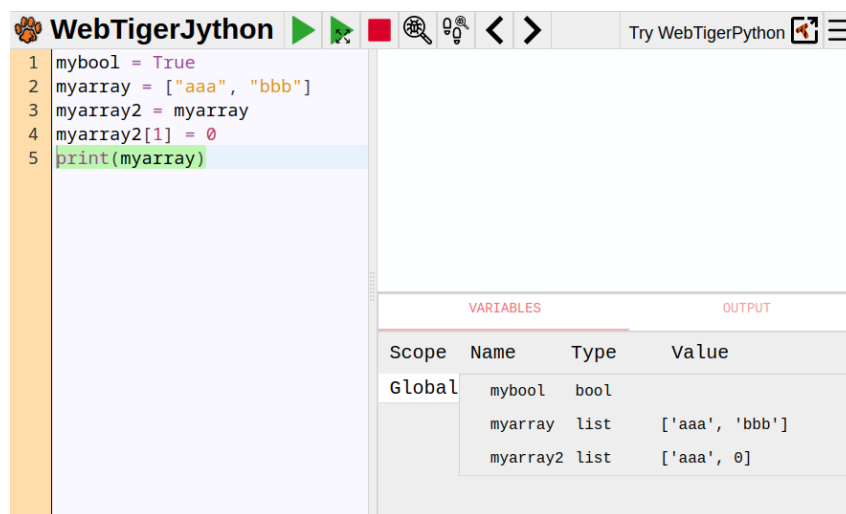


Figure 4.1. WebTigerJython's debugger.

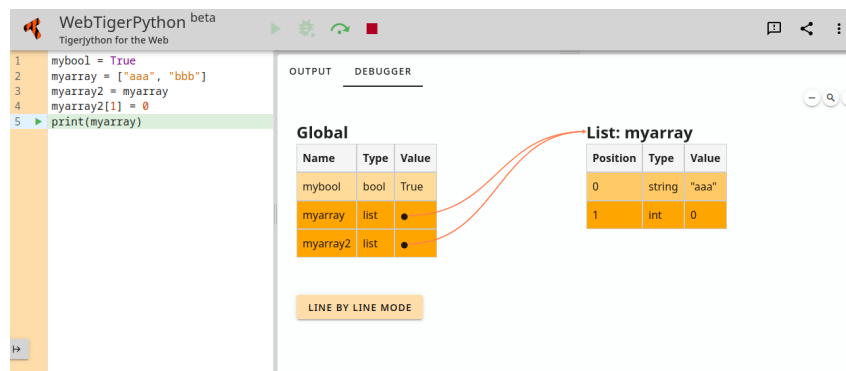


Figure 4.2. WebTigerPython's debugger for the same code snippet.

Chapter 5

Results

In this chapter, the current functionality of the debugger as well as the findings of the performance evaluation and the user study are discussed before leaving way to a short mention about methodological shortcomings.

5.1 Functionality

This section discusses the current functionality of the debugger's implementation, what it properly achieves doing and what are its shortcomings.

Current Functionality

The debugger clearly and accurately captures and displays the content of the memory for the following types:

- String
- Integer
- Float
- Boolean
- List
- Tuple
- Set
- Frozenset
- Dictionary

In particular, it correctly keeps track of shared references using arrows and highlights the last datastructures to have been changed in four levels, fading out after every new execution steps.

In most standard cases, executing the program line by line or breakpoint by breakpoint works. In particular, it also works in case of imports and user inputs as well as

user defined functions and the Turtle Graphics [15] feature. During the user testing and in particular during the free coding session, no errors or oddities were noted or disrupted the course of the study.

Current bugs/errors

There are a few edge cases which were discovered during an additional testing session after the user study which are yet to be addressed or improved on. In particular and non-exhaustively, the current debugger implementation will fail in the following cases:

- Bytearrays, imported types (from the `collections` library for example), `numpy` types, `pandas`, etc. are generally not displayed in a clearer way than Python's default string representation.
- The debugger doesn't work with the GPanel [30] feature.
- Custom classes are not always elegantly displayed.
- Using list comprehension to define a list will undesirably expose memory objects and addresses to the user.
- Functions within print statements are not always properly handled, stepped over or into, as for example in `'print(f"Fibonacci number for n is fibonacci(n))"'`.
- Objects in memory that are too big to be displayed will crash the program.
- Using `"#bp"` on an empty line will not work to set up a breakpoint.

Current limitations

The present implementation uses various tricks to make Python execution look simpler than it actually is. For instance, when initializing classes, we make sure the line gets executed “at once” though Python initializes in two steps. The debugger will not show the inner-workings of imported functions nor will for standard library ones. When it comes to memory state capture, a lot of the content of the memory is filtered out such that practically only user-defined variables and constants are displayed. While I believe that, in most cases, hiding some of the complexity will make using the debugger more intuitive and more helpful, not cluttering the user with information also means that, sometimes, some information needed to fix a problem will be missing.

Another important limitation of the current implementation is that the display of each “complex” datatype has to be handled separately. The current method of Memory State Capture is not general enough to allow arbitrary datatypes to be sent to the front-end in a way that can be properly understood and processed in the form of a table. Moreover, the form of the tables differs based on the datatype. For sets, there are only two columns as the order of elements does not play a role. For dictionaries, we need 4 columns since the type of the key and the type of its associated value might differ. In its current state, there is no way for the front-end

of the debugger to understand how to meaningfully render a datatype that has not been especially and individually handled. Imported types, for example from the `collections` [31] module, will in general not have proper readable associated tables for all their references.

An additional shortcoming of the current implementation is the display of nested structures as well as references within them. Indeed, currently, there is one column for simple stack variables and one column for detailed variables. If detailed variables themselves contain complicated structures, such as lists, sets, dictionaries, their string representation will simply be displayed in the value column. This likely makes the debugger less than optimal when it comes to the understanding of sophisticated datastructures. While the display of deep and/or nested datastructures comes with its own sets of challenges (circular dependencies, cross-references, etc.), major improvements are possible. For instance, the code snippet below ?? will throw a vast amount of "[]" parenthesis at the user surrounding "<unrepresentable object: maximum recursion depth exceeded while calling a Python object>" whereas Python Tutor handles it gracefully as displayed in Figure 5.1.

```

1 # Creating a circular reference between two lists
2 a = []
3 b = []
4
5 a.append(b) # a contains a reference to b
6 b.append(a) # b contains a reference to a
7
8 # Now a and b are in a circular relationship
9 print(a) # Output: [[...]] (b is inside a, and a is inside b)
10 print(b) # Output: [[[...]]] (a is inside b, and b is inside a)

```

Listing 5.1. Code snippet creating a datastructure with circular references

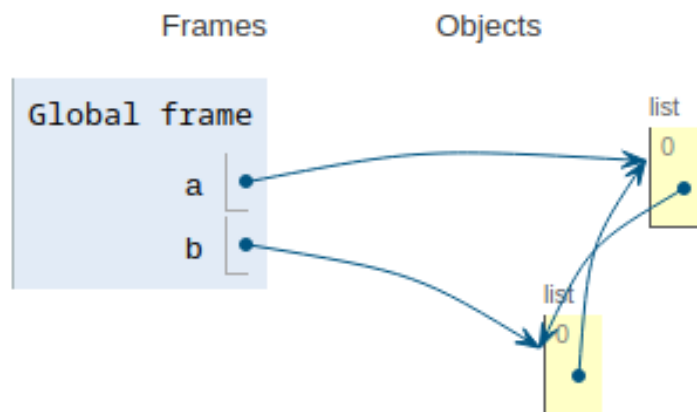


Figure 5.1. Python Tutor's handling of circular datastructures.

Other desired features that WebTigerPython's currently lack and have been mentioned during the evaluation process are the display of the return value of functions

before they return, the display of private and protected variables, the display of the memory state during instance initialization in object oriented programming projects, the display of class variables, a configurable depth of visualization, the ability to hide objects from the display.

5.2 Performance

The debugger significantly slows down the execution of programs containing a large number of Python operations. This is however only due to the part of the debugger that deals with altering the execution flow and not the memory state capture. Should it in the future be of interest to debug programs while they are running, a few simple changes should enable re-using the current technology. Considering the context and the environment surrounding WebTigerPython, it was assumed that performance was not of primary concern and, in the case that it should be, it is likely that one would use specialized methods implemented directly in lower-level languages, the performance of which the debugger doesn't significantly impact.

Furthermore, the addition of the debugger to WebTigerPython does not substantially impact its loading time.

5.3 User Study

The study was conducted on 6 students and Python learners, four of which described themselves as "familiar users", one as a "novice" and one as a "confident user" as defined in [Section 4.2](#). They haven all been chosen for knowing some Python's basics while not being experts and have all voluntarily spent between 40 and 80 minutes following the protocol as described in [Section 4.2](#).

Guess Exercises

All participants have easily understood the task and were able to solve the first example puzzle without any hindrance. For the rest of the exercises, it is remarkable that when an incorrect answer was given, it was always the predicted "intuitive answer" described in [Section 4.2](#).

Different Execution Scopes To my surprise, only one out of the 6 participants got this exercise right immediately. Three however understood their mistake immediately upon revealing that their answer was incorrect. For the last two, a short lonely exploration with the debugger was enough to understand the code snippet. In this case, seeing the same variable name in different execution scopes or stack frames was a helpful feature.

Out of Range In this case again, only one participant figured the solution out within the first minute. Three participants needed the help of the debugger and the last two needed the help of the debugger and of additional explanations. It turned out however that the two main factors of confusion in this snippet were the starting/ending point of the `range()` function as well as the `numbers.pop(num)`

line. While in a lot of cases the debugger could help understanding what they do, a simple internet search or documentation search (which were not allowed during the experiment) or answer to a question about these two utilities would likely have been more helpful. In order to properly determine whether the current implementation of the debugger helps in detecting and fixing out of range error, an example with less confusion factors should be constructed.

Mutable Default Argument This exercise was more difficult and none of the participants could figure it out on their own. One participant, the more confident user, could understand what happened using the debugger alone but four needed additional explanation. It turns out that none of them knew that Python lists were mutable, or had a clear conceptual understanding of the difference between mutable and immutable objects. While the current implementation of the debugger could not provide such understanding on its own, it is my experience that the display using arrows was a helpful visual support to help them realize what the correct answer was and why.

Off by One The results for this exercise were very split. On one side, two participants immediately understood the correct result, one needed the debugger, two needed additional explanations. Surprisingly, one of them got the right answer for wrong reasons; in this case, assuming the first element would not be included in the count (while in reality it is the last elements which was not included in the count). My hypothesis is that the understanding of this code snippet was very reliant on the participants precise knowledge of the `range()` function, which in some cases was influenced by the explanations provided when discussing the out of range example. In this case, it is worth noting that most participants who got the wrong answer did not have the patience to carefully step through the function with the debugger indicating either a certain clumsiness of the system or some redundancy or lack of interest induced by the form of the exercise. Here again it is to be remarked that the debugger offered a very helpful visual support for the explanation.

Mutables and Immutables This exercise is very similar to the mutable default argument one and got very similar though better participant performance. For those who had the conceptual explanation of what mutable were during it performed better on this one and for those who still had not heard of it during this exercise performed better during the other (the order being randomized). The debugger could alone help those who had a fresh understanding of the concept of mutable objects while providing a useful platform for the explanations otherwise.

Unexpected Collisions Finally, this last code snippet was probably too highly artificially created to demonstrate the effectiveness of the debugger by leveraging a rare bug whose functioning is comprehensively captured in the debugger's display. Nevertheless, it is gratifying to see that all participants, after an initial mistake, understood the code snippet alone with the debugger and did not require additional explanations.

Free Exercises

In total, 3 participants agreed to use the debugger in a somewhat more liberated coding exercise intended to observe the use of it in less artificial conditions. Two of them coded the Fibonacci sequence, as suggested. One of them did not use the debugger and solved the exercise in about 8 minutes while the other needed 13 and used the debugger mostly in order to solve off-by-one errors. The third one imported one of their own coding project in the web interface, in this case a timed mental arithmetic test, and started to play around. They were able to gain additional understanding of how the `time` library and function work as well as fix a bug carefully stepping into one of the functions and observing the content of a list at a given moment in time.

While the experience with the debugger was mostly neutral or positive, the participants constantly switched between the output console and the debugger view probably indicating a need to enable a side-by-side view. Moreover, when importing a long (about 300-400 lines) coding project in the interface, it became very difficult, during the step by step execution, to figure out which line was executing as the small green triangle turned out difficult to spot. An automatic scroll to the right position might be helpful in this case. Finally, it is amusing to note that the younger participants (15-18 years of age) seemed to enjoy playing with the arrows and their dynamic display, providing a pleasant experience but in one case distracting them from the task.

System Usability Scale

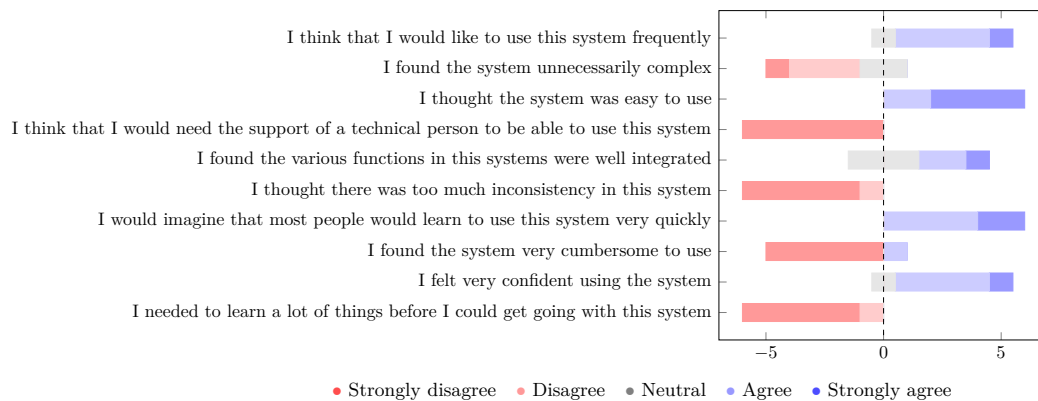


Figure 5.2. Accumulated answers of each item of the *System Usability Scale* questionnaire

The average system usability score across all participants is of 84.2 with a standard deviation of 8.5 generally indicating excellent usability [32]. However, while it was specified that in this case “system” should only refer to the debugger, it is not certain that the general experience participants were invited to note was not largely influenced by the navigation of the system as a whole. Isolating the testing of the debugger may not have been properly made possible through the standard use of the system usability scale.

Accompanying the questions above, there were 3 more specific questions about the participants' experience during the experiment asked, the results of which are displayed in Figure 5.3. They seem to indicate a generally positive experience with the debugger during the study.

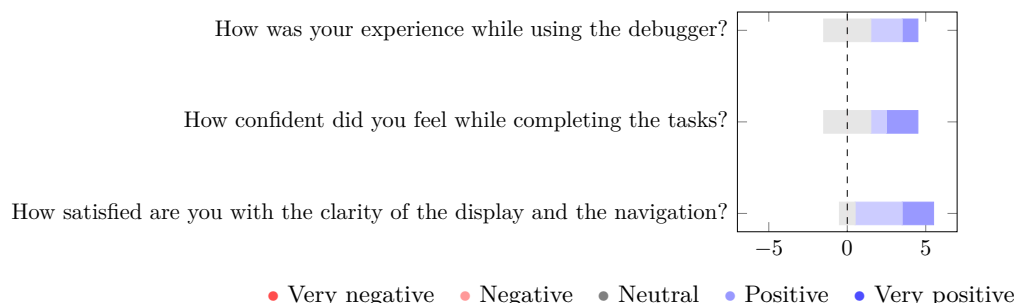


Figure 5.3. Accumulated answers of each item of the *Participant's experience* questionnaire

5.4 Methodological shortcomings

Functionality At the time of writing, there has been no large-scale nor systematic testing of the debugger's functionalities. While the core features of the debugger should work in most and usual cases, it is likely that more and new errors and display oddities will surface.

Performance When it comes to performance, the claim that there should be no significant performance impact when considering the memory state capture independently from the control flow modification makes logical sense but has not been properly tested considering the current architecture and should, at this point, be considered a likely hypothesis.

User Study The user study has been conducted on a small number of participants who all knew me. It is conceivable that some answers, particularly to the **System usability Scale** questionnaire, might have been biased by my presence and the knowledge that they were evaluating my work and that I would have a look at how they would evaluate it. Furthermore, the guess exercises were likely too difficult for most participants and do not offer a very genuine framework for their interaction with the debugger. It remains that most of them, four out of the six participants, report not using a debugger at all when coding.

Chapter 6

Conclusion

This section sums up the pivotal information of this work before offering a perspective into possible future work directions.

6.1 Key Findings

WebTigerPython, developed by the Center for Informatics Education at ETH Zürich, is a web-based Python learning environment. As such, adding a debugger to it, understood as a program or collection of tools to modify the execution environment of a program to enable interacting with it in additional ways in order to better understand its functioning is crucial. After a thorough exploration of the utilities provided in similar projects and considering the general intention of offering a minimal and clear display for novice users of WebTigerPython, it was decided to focus on three core functionalities:

1. The modification of the execution flow of the target program to allow halting it after every executed line, alternatively before every breakpoint.
2. The display of a selected content of the memory, namely user-defined variables, datastructures and classes.
3. The display of relationships between and the execution context of this selected view of the memory.

Halting the execution of the program at selected points was implemented using Python's `bdb` module. It involves mostly overriding the default behavior of line execution, function return, function call and exception handling. Capturing the state of the memory was done using Python's `dis` module. In both cases, offering a seamless user experience involved hiding Python's complexity through various tweaks of the original behavior.

It was chosen to render this information in two columns. The left column represents all variables in their stack frame in table fashion. For most, their value is directly represented but for most complex ones like lists, sets, dictionaries, etc., instead of a value, an arrow points to their detailed representation in the right column for an intuitive representation of shared references to the same object.

When it comes to the functionality, while the debugger is yet to be extensively tested, it seems to reach its goals and be particularly suited for short programs and relatively simple datastructures.

Performance-wise, programs intensely relying on a high number of Python operations, that is, operations not optimized or implemented in lower level languages are heavily penalized by the use of the debugger and are slowed down by a factor between 60 and 100. This is due to the part of the debugger that is aimed at stopping the execution at every line/breakpoint. Memory state capture can, in need, run independently.

The user study seems to reveal excellent usability and ease of navigation. The tool seems very intuitive to use. Over-generalizing a little one could claim that it appears to be helping students to understand, find and fix coding errors related to concepts they already know and can offer a helpful visual support for explainers to introduce new ones.

A comparative evaluation of WebTigerPython's debugger with its predecessor indicates significant improvement in numerous major areas.

6.2 Future Work

The debugger could be further enhanced implementing various small changes and fixes listed in [Section 5.1](#). On a more fundamental level, the following functionalities could probably be added to the debugger and improve it significantly without betraying the spirit of simplicity that has guided its development:

- Adding the ability to step into lines of code into the details of the execution of expressions.
- Adding the ability to step over and out of functions automatically, that is, without the need of explicitly defining breakpoints.
- Adding explicit support for common types used in the `collections` library, `numpy` objects, `pandas` and `Gpanel`.
- Improving on the display of complex data structures, for example by adding a configurable amount of columns and implementing a more flexible arrow display.
- Various improvements can be brought to the navigation and usability, in particular by enabling side-by-side display of the output console and the debugger as well as automatic scrolling in the code view to the currently executing line of code.
- Control flow visualization could be added to the debugger's functionalities.

A proper testing of the tool would probably do it a lot of good. On a more fundamental level, it is not clear how much WebTigerPython's users will engage with the debugger, to what extent it will answer their needs and even, what those needs precisely are.

For the moment, the tool seems to be living up to its modest promise but there's still so much to improve and explore!

Bibliography

- [1] Clemens Bachmann. WebTigerJython 3 A Web-Based Python IDE Supporting Educational Robotics. Master's thesis, ETH Zurich, 2023.
- [2] Nicole Trachsler. Webtigerjython - a browser-based programming ide for education. Master thesis, ETH Zurich, Zurich, 2018.
- [3] Tigerjython. <https://tigerjython.com/en>. Accessed: 2024-24-09.
- [4] Python tutor. <https://pythontutor.com/>. Accessed: 2024-24-09.
- [5] Thonny Integrated Development Environment. <https://thonny.org/>. Accessed: 2024-24-09.
- [6] Tom Britton, Lisa Jeng, Graham Carver, Tomer Katzenellenbogen, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". 11 2020.
- [7] W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent catastrophic accidents: Investigating how software was responsible. pages 14–22, 2010.
- [8] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures*, 44:89–113, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).
- [9] IBM. What is debugging? <https://www.ibm.com/topics/debugging>. Accessed: 2024-24-09.
- [10] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, New York, 1996.
- [11] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. page 572–583, 2018.
- [12] ETH Zurich. Webtigerjython. <https://webtigerjython.ethz.ch/>. Accessed: 2024-09-24.
- [13] Unknown. Python tutor unsupported features. https://docs.google.com/document/d/13_Bc-12FKMgwPx4dZb0sv7eMfYMhRVgBRShha8kgbU/edit. Accessed: 2024-09-24.

- [14] Python Software Foundation. bdb — debugger framework — python 3.12.0 documentation. <https://docs.python.org/3/library/bdb.html>. Accessed: 2024-09-24.
- [15] Python Software Foundation. turtle — turtle graphics — python 3.12.0 documentation. <https://docs.python.org/3/library/turtle.html>. Accessed: 2024-09-24.
- [16] Julia Bogdan. Turtle Graphics for WebTigerJython-3. Bachelor’s thesis, ETH Zürich, 2023.
- [17] Python Software Foundation. pdb — the python debugger — python 3.12.0 documentation. <https://docs.python.org/3/library/pdb.html>. Accessed: 2024-09-24.
- [18] Liran Haimovitch. How python debuggers work - pybay2019. YouTube. Presented at PyBay2019 - 4th annual Bay Area Regional Python conference. Accessed: 2024-09-24.
- [19] Python Software Foundation. inspect — inspect live objects — python 3.12.0 documentation. <https://docs.python.org/3/library/inspect.html>. Accessed: 2024-09-24.
- [20] Python Software Foundation. dis — disassembler for python bytecode — python 3.12.0 documentation. <https://docs.python.org/3/library/dis.html>. Accessed: 2024-09-24.
- [21] Mark Lutz. *Learning Python*. O’Reilly Media, Sebastopol, CA, 5th edition, 2013.
- [22] Evan You et al. Vue.js. <https://vuejs.org>, 2014. Accessed 2024-02-08.
- [23] Eduardo San Martin Morote and the Pinia Contributors. Pinia - the intuitive store for vue.js. <https://pinia.vuejs.org/>. Accessed: 2024-09-24.
- [24] The Pyodide development team. pyodide/pyodide. <https://doi.org/10.5281/zenodo.7570138>, August 2021.
- [25] John-David Dalton and the Lodash Contributors. Lodash: A modern javascript utility library delivering modularity, performance & extras. <https://lodash.com/>. Accessed: 2024-09-24.
- [26] Anatoos. Leaderline: Draw a leader line in your web page. <https://anseki.github.io/leader-line/>. Accessed: 2024-09-24.
- [27] NumPy Developers. numpy.linalg — numpy v1.25 manual. <https://numpy.org/doc/stable/reference/routines.linalg.html#module-numpy.linalg>. Accessed: 2024-09-24.
- [28] Python Software Foundation. Cpython: listobject.c — implementation of python lists. <https://github.com/python/cpython/blob/main/Objects/listobject.c>. Accessed: 2024-09-24.

- [29] John Brooke. Sus: a "quick and dirty" usability scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland, editors, *Usability Evaluation in Industry*. Taylor and Francis, London, 1996.
- [30] Python-Online. Gpanel - python graphics library. https://www.python-online.ch/index.php?inhalt_links=gpanel/navigation.inc.php&inhalt_mitte=gpanel/gpanel.inc.php. Accessed: 2024-09-24.
- [31] Python Software Foundation. collections — container datatypes — python 3.12.0 documentation. <https://docs.python.org/3/library/collections.html>. Accessed: 2024-09-24.
- [32] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the system usability scale (sus). *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008.

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden drei Optionen ist in Absprache mit der verantwortlichen Betreuungsperson verbindlich auszuwählen:

- ☒ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz¹ verwendet.
- ☐ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz² verwendet und gekennzeichnet.
- ☐ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz³ verwendet. Der Einsatz wurde, in Absprache mit der Betreuungsperson, nicht gekennzeichnet.

Titel der Arbeit:

Building a Debugger for WebTigerPython

Verfasst von:

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):

Stijve

Vorname(n):

Theo

Ich bestätige mit meiner Unterschrift:

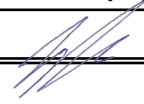
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum

Zürich, den 25. September 2024

Unterschrift(en)



Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

¹ z. B. ChatGPT, DALL E 2, Google Bard

² z. B. ChatGPT, DALL E 2, Google Bard

³ z. B. ChatGPT, DALL E 2, Google Bard