

UNIVERSITY OF TARTU
Institute of Computer Science
Informatics Curriculum

Alar Leemet

Omniscient Debugger for Thonny Integrated Development Environment

Bachelor's Thesis (9 ECTS)

Supervisor:
Aivar Annamaa

Tartu 2018

Omniscient Debugger for Thonny Integrated Development Environment

Abstract:

Thonny is an integrated development environment for Python 3 programming language, designed for beginner programmers. This thesis aims to document Thonny's back-end, debugger and improve Thonny's debugger by giving it the ability to display previous program states, called omniscient debugging. The reader is first introduced to common debugging techniques. Then an overview of Thonny and its current version's functionalities is given. After that, the architecture of Thonny's current debugger, the structure and the implementation of the omniscient debugger are described. Finally, the preliminary beta testing results are presented.

Keywords:

Python, debugging, IDE, omniscient debugger

CERCS: P175, Informatics, systems theory

Kõiketeadev silur arenduskeskkonnale Thonny

Lühikokkuvõte:

Thonny on Python 3 programmeerimiskeele integreeritud arenduskeskkond algajatele programmeerijatele. Bakalaureusetöö eesmärk on dokumenteerida Thonny tagasüsteemi ning silurit ja täiendada Thonny silurit, andes sellele programmi eelmiste olekute kuvamise võimaluse, mida kutsutakse kõiketeadvaks silumiseks. Esmalt tutvustatakse lugejale levinumaid silumistehnikaid. Seejärel antakse ülevaade Thonnyst ja tema hetke versiooni funktsionaalsustest. Siis kirjeldatakse Thonny praeguse siluri arhitektuuri, teostatava kõiketeadeva siluri arhitektuuri ja elluviimist. Lõpuks esitatakse esialgsed beetatestimise tulemused.

Võtmesõnad:

Python, silumine, IDE, kõiketeadev silur

CERCS: P175, Informaatika, süsteemiteooria

Table of Contents

Introduction	4
1 Debugging	5
1.1 Conventional Debugging.....	5
1.2 Reverse Debugging	7
2 Thonny	10
2.1 Overview	10
2.2 Features of Thonny.....	11
2.2.1 Different Views in Thonny	11
2.2.2 Debugger of Thonny	13
3 Documenting the Back-end of Thonny	17
3.1 Main Logic	17
3.2 Debugging Logic	18
4 Implementation of Omniscient Debugging	21
4.1 Initial Attempt	22
4.2 Interim Solutions	22
4.3 Final Solution	24
5 Testing of Thonny’s Omniscient Debugger	28
5.1 Opinion Poll on Thonny’s Debugger	28
5.2 Testing Memory Usage of Thonny.....	28
6 Conclusions.....	31
7 References	32
Appendix	35
I. Testing Resources	35
II. License.....	50

Introduction

The information technology industry requires more qualified staff with each passing year. To satisfy the increasing demand, software developers need to be trained ever faster. However, the increasing speed of teaching must not compromise the novice programmers understanding of programming's basic concepts. Hence tools for learning programming like Thonny exist, illustrating the core ideas for the beginners to comprehend. Understanding the basics very well is an important prerequisite for a novice developer to become the specialist the industry needs.

The idea of the author of Thonny, Aivar Annamaa, was to create a Python 3 integrated development environment for visualizing code execution for beginner programmers. The visualization is mainly handled by the debugger of Thonny, allowing the user to step through the running program's code with the granularity of sub-expression evaluation. The debugger can be enhanced to make the debugging process faster and more convenient for the novice developer.

Consider a beginner programmer trying to find a bug in a written program. The developer will at some point use debugging tools, starting to step through the program state by state. By the time the general location of the bug is determined, the developer may have accidentally stepped over the interesting piece of code or forgotten what the previous state displayed several times. Both cases necessitate the restart of the debugging session, which may become frustrating very quickly for novice developers. To make the experience smoother, stepping back in time could be made available.

This thesis describes an open source software development project that aimed to document the structure of Thonny's back-end, debugger and further improve Thonny's debugger. Development is done by altering the existing debugging infrastructure so that the debugger gains the ability to display past program states. This capability is called omniscient debugging. The feature helps beginner programmers to quickly play and replay code execution by stepping forward and backward in saved program states without restarting the debugger.

The thesis will first describe common debugging methods based on existing literature on the subject. Then an introduction to Thonny will be given, describing its purpose and most important features. An overview of the existing architecture of Thonny's back-end and debugger will be presented, upon which the description of the development and the final solution follows. Finally, the preliminary results from Thonny's beta testing will be provided.

1 Debugging

An important part of software development is the practice of debugging software. It is a process of identifying a problem, locating it in code and either correcting or determining workaround for that problem [1].

Debugging is mainly done in three different stages of software development. The first is the initial development, where the specification is translated into code – errors made by the programmer must be debugged before moving on to later stages of development. The second is testing in later stages, where the complete solution is checked. Debugging is necessary to determine if an unexpected behaviour is caused by a bug or a faulty test case, which may in turn be caused by inconsistencies in software specification. If the fault is caused by the program, further debugging is needed for locating and fixing the bug. The third place is after the deployment of the software – debugging may be required to address poor performance under load or inadequate recovery from a failure [2].

To make this process more convenient for developers, many programming languages possess architectures for debugging code and many integrated development environments (IDEs) have front-end implementations for various debuggers built in [3]. Using those tools may also help a novice programmer to learn basic concepts of programming – stepping through the code can visualize principles like control flow, expression evaluation, function calls, recursion and shared mutable data [4]. Despite the available tools, on average 50% of developers programming time is spent on finding and fixing bugs [5]. A 2002 U.S NIST study suggests that almost 80% of all development cost is spent on identifying and correcting defects [6]. This section aims to give a brief overview of some debugging techniques and their properties.

1.1 Conventional Debugging

There are numerous ways of debugging software with various amounts of prerequisites needed before starting the process. Some of these require specific software in place, others the expertise of the developer. The following methods are based on forward execution of the program.

The first technique is log based debugging (also called print and peruse [7] or print debugging). Using this method, programmers insert logging statements (for example, *print* functions) directly into the source code and run the program, sending the desired data – be it

variables or function results – to standard output or into a file for investigation [8]. To make use of this technique, the user must be able to guess the general location of the bug to start inserting logging statements and closing in to the core of the failure. This approach requires at least a simple text editor for writing logging statements directly into the source code of the program and an interpreter for running the code. There are some disadvantages to this approach though. First of which is the requirement to modify the source code either by implementing a logging architecture or inserting *print* statements and undoing the changes afterwards. Also, the user should be able to balance the amount of logging statements used: too many and the user cannot distinguish the important from the noise, too few and the critical information may not be displayed. This technique also relies heavily on running the program. If the user cannot guess the possible source of the failure and running the program takes a large amount of time, debugging the program may quickly become frustrating. Thus, this method is more suited for beginners debugging smaller programs or platforms where other tools are unavailable [7].

The second way to debug programs is to use debugging tools, either standalone or bundled with IDE-s. Most of these debuggers work with break points – locations in code, where the execution of the program is halted once code execution reaches it. The user can choose where to insert those points. Upon reaching a break point, the tools usually display the current state of the program – values stored in variables, execution stack etc. Afterwards, the user can choose to continue program execution, either by stepping – executing the next statement(s) – or by finishing program execution by halting the debugging process or stepping to the end. This method allows the user to slow down the execution of the program and check different states of the running software. There are some shortcomings though. One can only use this method if appropriate standalone or bundled tools for the platform are available. This may not be the case if development is carried out in embedded environments [7]. Also, information about the program states is limited only to the current call stack. This means that previous states are only accessible by re-running the program. A routine of setting breakpoints and running the program is initiated, which continues until the bug has been detected [8].

Different debugging approaches give the developer the ability to check the states of interest or slow down the entire execution process of the program at hand. These capabilities may help the developer to track down encountered issues. The choice of technique should be

made according to the failure encountered, development platform and skills of the developer. As the currently discussed techniques often rely on time consuming repetitive routines, more efficient approaches are needed.

1.2 Reverse Debugging

The techniques in the previous section were both based on forward execution of the program under scrutiny. This means that during the debugging process, the program's code is always run in its intended direction. These methods are cyclic debugging techniques [9]. The term comes from the fact that the erroneous program is run and rerun many times, during which the developer checks variables, output or sets new breakpoints after each run to pinpoint the issue. This forms a cycle of running the program and checking its state, from which the term's name originates. Cyclic debugging methods have a common weakness – they do not work well with interactive and/or non-deterministic programs. Debugging asynchronous programs with cyclic debugging methods may affect timings of parallel processes, making bugs irreproducible or cause additional failures, which may not occur during normal execution [9]. In addition to cyclic debugging methods, there are other approaches to debugging, which have a different philosophy underneath.

To counter issues cyclic debugging may have left unaccounted for, reversible debugging techniques are available. The idea of reversible debugging is to tackle the issue inside the run that has already failed instead of trying to reproduce the issue in a separate run [9]. To achieve this, backwards stepping capabilities are added to the debugger that function alike stepping forward in a traditional debugger but backwards in time. This will increase the efficiency of the tool – instead of starting a new debugging session each time when encountering an issue or realizing the issue is before the observed state, the user can now instantly step back through the program's past states to the point of the bug's occurrence [10].

As it is impossible to truly reverse the execution of programs because of machine instructions that destroy information, data must be saved for later reconstruction of past states. There are two fundamental technical concepts for returning to past states. The first of which is to record all the information needed to display the past state. The second option is to reconstruct the state by continuing execution from a saved checkpoint [9].

The first reversible debugging approach is called omniscient debugging (also known as record-replay debugging [9] or history logging [10]) [8]. This technique bridges the gap between cyclic and reversible debugging. It is based on saving entire previous states of the

program for later display. This gives the ability to cyclically debug a non-deterministic program and is easier to implement, as this solution does not require many changes to the existing debugger architecture [9]. The technique combines the advantages of both discussed cyclic debugging techniques: saving of program state logs and step-by-step execution both forward and backward [8]. Omniscient debugging cannot be considered as true reverse debugging though as the program is not rerun when querying past states but previously saved states are displayed [11].

Another solution would be a reconstruction or re-execution based debugger. It operates by saving specific points in the program execution that can be reliably reconstructed and returned to. This is called checkpointing. Those checkpoints are later used for running to a breakpoint in a past state. To run backwards to a breakpoint, the debugger starts running through the states that are after the checkpoints – a checkpoint is loaded, then the program is run forward. When a breakpoint is reached, the debugger runs again to its current state, loads the checkpoint and stops at the breakpoint. Additionally, solutions for handling input/output (I/O), thread interactions, file system operations and other communications with external resources must be considered. Lastly, sufficient control over the target system is needed for it to repeat its execution. For this, standard system libraries, language virtual machines or virtual platforms may provide the necessary tools and flexibility [9].

Both discussed reversible debugging approaches have their strengths and weaknesses. As omniscient debugging records the entire execution of the program, it makes it easier to browse the states of the erroneous execution and search for the bug that caused the failure [8]. Re-executing debuggers on the other hand makes it possible to see the computation unfold (for example stepping back and over random number generating statements returns a new number) and makes the program capable of reacting to different inputs while debugging [11]. Both techniques produce overhead while handling debugging. For omniscient debugging, the necessity to save large amounts of data about program states and the saving process itself could produce high overhead while debugging. For re-executing debugger, overhead is produced by the re-execution itself, which may not be as large as with omniscient debugging. However, other difficulties for re-execution come with locating the desired point while executing forward from a past state and ensuring the program is executed the exact same way each time [10]. These are most likely the reasons why reversible debugging is not common among development tools.

Reversible debugging provides solutions to debug programs that were previously considered to be not efficiently debuggable. It also makes debugging more convenient by providing means to go backward in time to check states that may have been skipped but turn out to be important after all without the need to rerun the program [7]. This may save time spent on debugging. A study concluded that developers using reversible debuggers spent an average of 26% less time on debugging than developers using traditional, forward-executing debuggers [5]. Resulting time savings may help decrease costs in software development in general.

2 Thonny

This chapter will first give an overview of Thonny – the software for which an improved debugger is developed in this thesis. After that, the most important features of Thonny are described. Then, a more detailed insight into Thonny’s debugger is provided. The chapter is a summary of the Thonny overview chapter from the bachelor’s thesis of Taavi Ilp [12], Aivar Annamaa’s articles on Thonny [4,13] and Thonny’s homepage [14].

2.1 Overview

Thonny is an integrated development environment (IDE) for Python 3 programming language for novice programmers. Thonny was developed by Aivar Annamaa at the University of Tartu (UT) and is being used in introductory programming courses at the UT. Thonny is also bundled in Raspbian OS as a substitution for Python IDLE, making it easier for beginners to learn coding while being useful to experienced Python programmers as well [15]. Thonny is available for Windows, Linux and Mac OS platforms and comes with built-in Python 3.6 interpreter to allow novice developers to instantly start programming after installation. It is open-source and free to use, the source code being available at a public Bitbucket repository [16] under the MIT license. As of 12.05.2018, Thonny version 2.1.16 has been downloaded a total of 33302 times across all platforms (excluding Raspberry Pi users). Figure 1 displays Thonny’s user interface with a simple program, that has been executed.

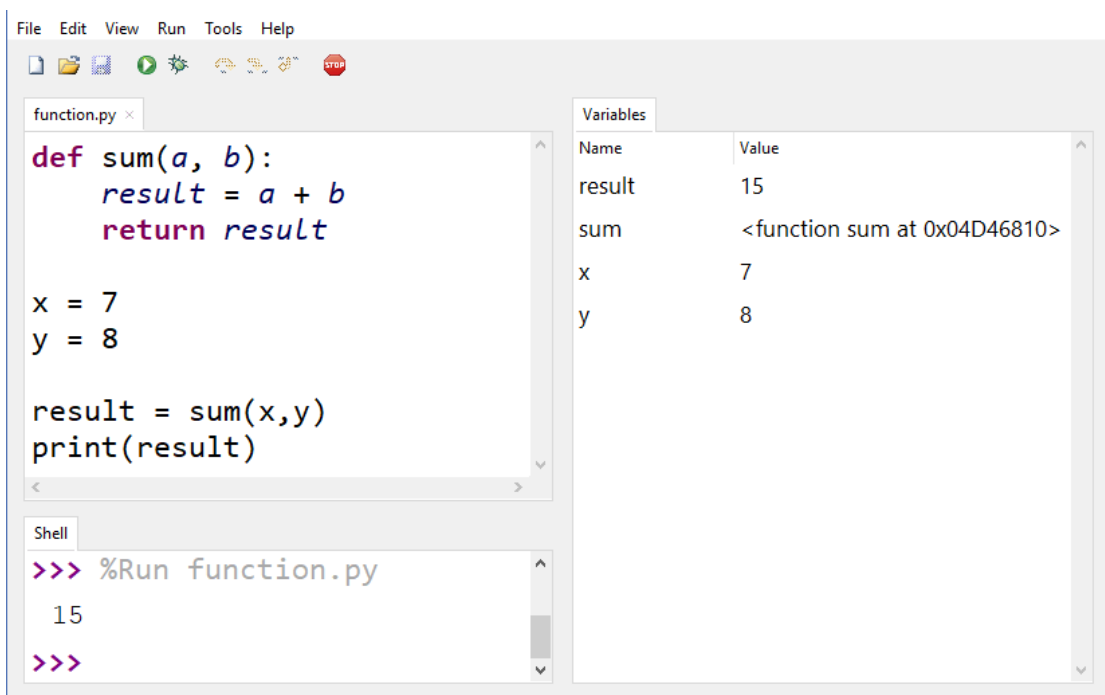


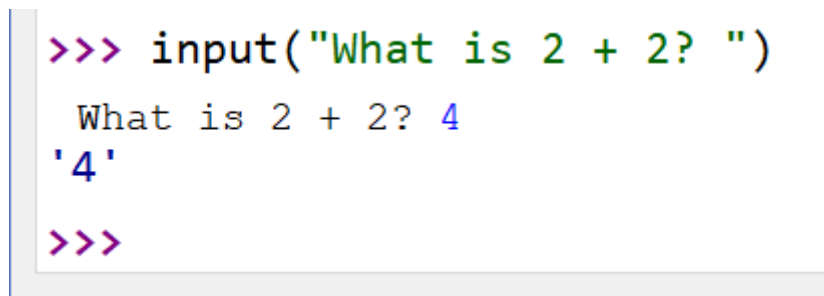
Figure 1. User interface of Thonny IDE.

2.2 Features of Thonny

Thonny has features aimed at helping novice programmers understand concepts of programming. The most prominent of those are the ability to step through the code, step-by-step expression evaluation, visualization of the call stack and modes for explaining the concepts of references and heap. The following features can be seen in use by referring to Figure 1.

The main functionality of any IDE is the editor. Thonny's editor offers core IDE features like syntax colouring, parentheses matching, code completion, automatic indentation, block indentation and block commenting. It also features the ability to open multiple files under different tabs. Syntax error highlighting for open parentheses and quotation marks is also available.

Thonny also provides a shell, which is an enhanced version of Python's IDLE shell. In contrast to Python's IDLE, Thonny's shell is integrated to the main window as this makes using the shell more comfortable for the novice programmer. The shell also uses different formatting for program input/output and shell operations to separate different shell events more clearly. This can be seen in Figure 2 – expression output is coloured black; expression input is blue and expression evaluation result is in bold and dark blue.



```
>>> input("What is 2 + 2? ")
What is 2 + 2? 4
'4'
>>>
```

Figure 2. Different formatting for input, output and expression evaluation in shell.

2.2.1 Different Views in Thonny

In addition to basic IDE features, Thonny has many additional views available that are initially hidden from the user and can be opened on demand. These can be accessed by opening them via the “View” dropdown menu. These views are meant to visualize basic concepts of programming for novice developers.

To demonstrate the concept of variables, Thonny's “Variables” view can be opened by selecting “Variables” from the “View” dropdown menu. Then a table with two columns appears on the right side of the window. The first column displays the names and the second

the corresponding values of the variables. The table is updated automatically in accordance to the program being executed.

For illustrating the concept of references, the “heap” box can be opened similarly from the “View” dropdown menu. As seen on Figure 3, a table appears on the lower right corner of the window, displaying memory addresses in use by the program and the values saved in those addresses. If the variables table is also open, values in the variables table are replaced with the values’ memory addresses as displayed in the heap table. Additionally, when evaluating expressions in shell, the value’s ID is displayed as the evaluation result instead of the value itself. The changes introduced by “Heap” should help beginners understand how references affect code execution.

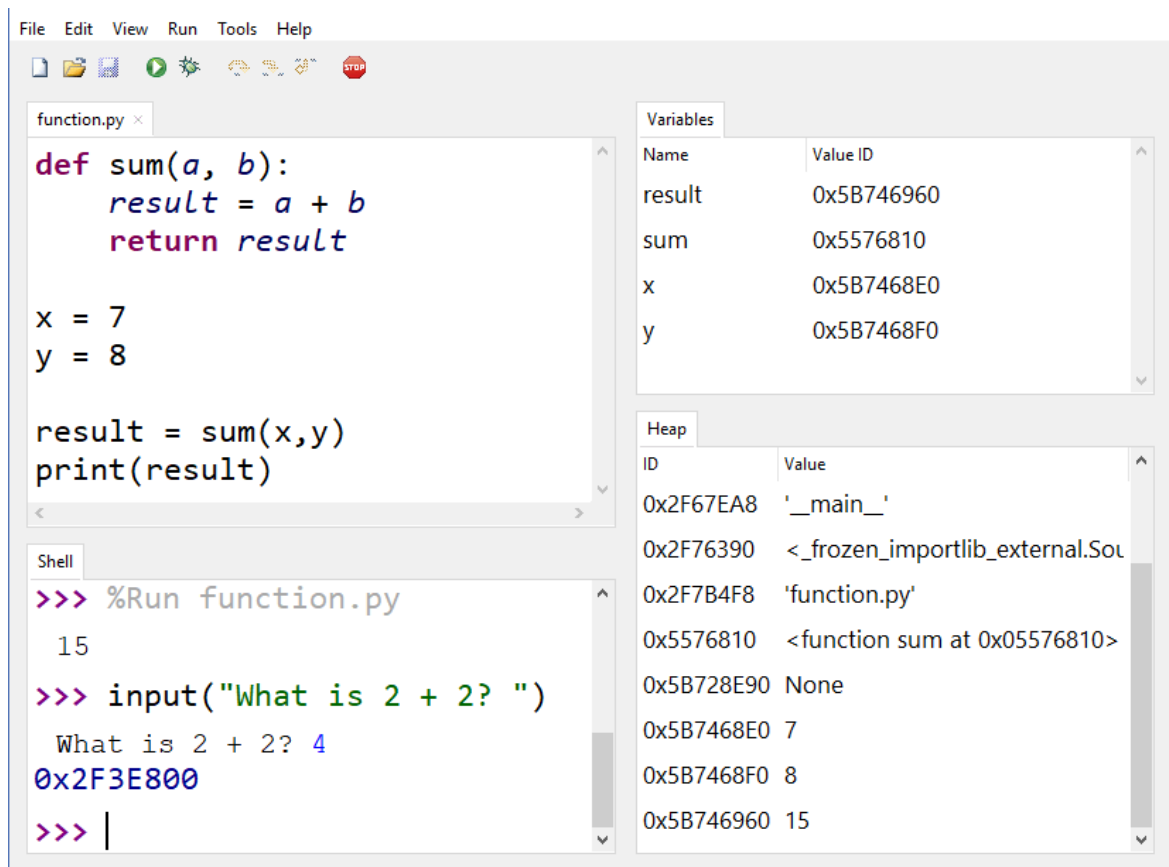


Figure 3. Thonny’s user interface with “Heap” view open.

To better describe objects used in the developer’s code, “Object inspector” box can be opened from the “View” menu. The initially empty box is populated with data by selecting an object in “Heap” or “Variables” view as seen on Figure 4. The first three lines of data in the box describe the object’s ID, its value and type. Then, depending on the selected object’s

type, a box displaying different details of the object may follow. This applies to the next types:

- For strings, a text box showing the content of the string is displayed.
- For collections, a table with the elements of the collection are displayed.
- For functions, the source code of the function is displayed.
- For text files, the content of the file and the current position of the pointer in the file is displayed, explaining how many lines and symbols have been read from the file.

Lastly, the attributes of the object are displayed.

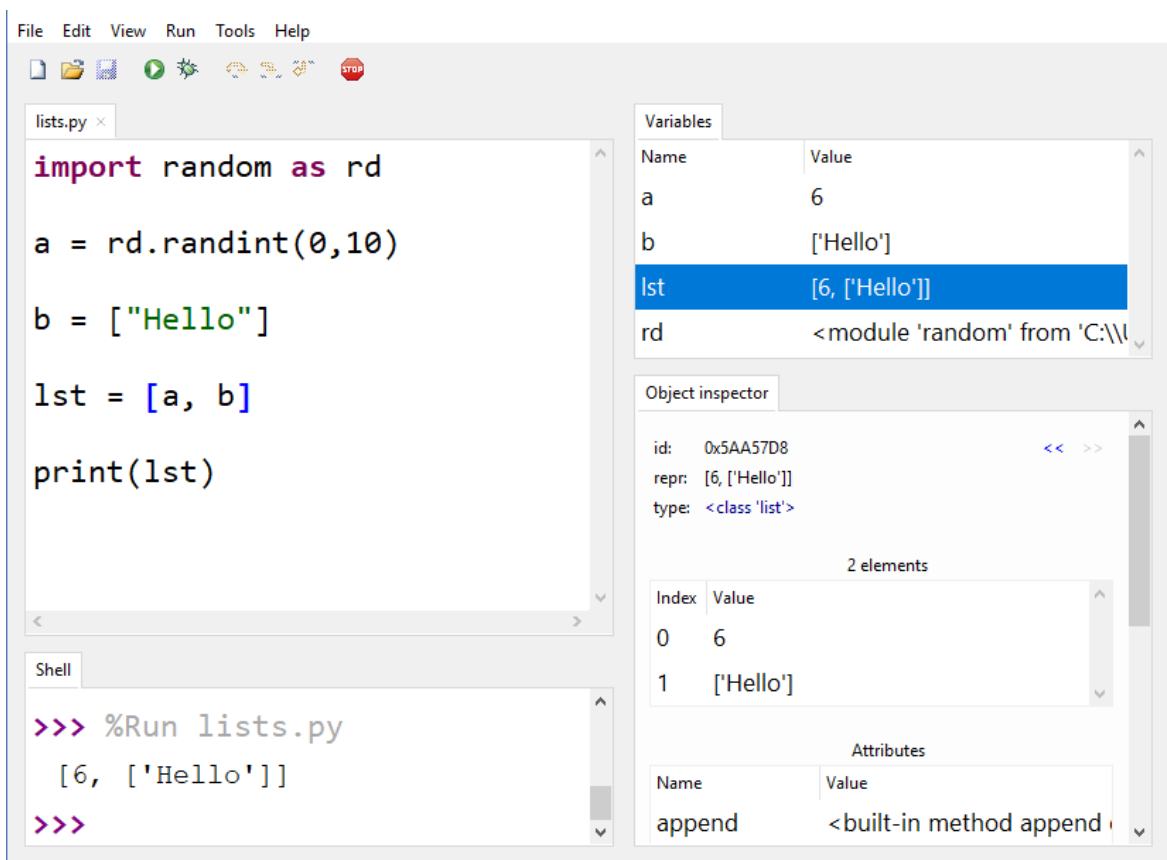


Figure 4. Thonny user interface with “Variables” and “Object inspector” open.

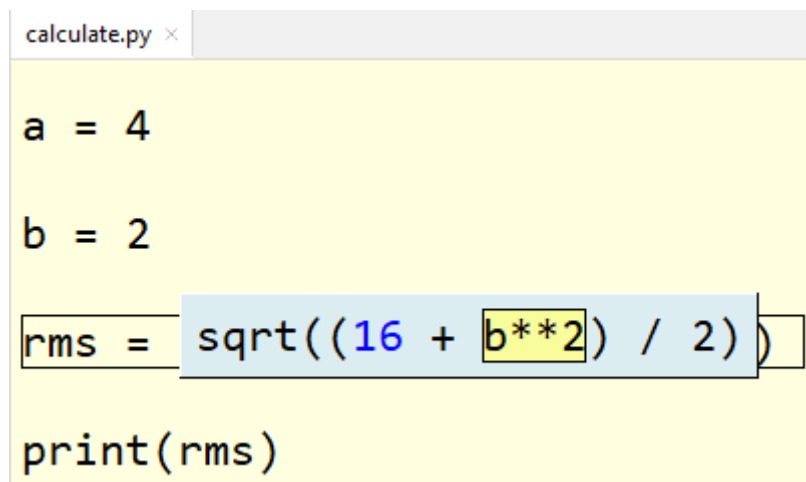
The described features provide the beginner developer options for better observation of program behaviour. All of them can be enabled or disabled on demand, giving more space for the advanced developer or less clutter for the student writing the first “Hello world!”.

2.2.2 Debugger of Thonny

Thonny’s most prominent feature is its debugger. It provides the ability to execute the user’s program step by step, helping the user understand the basics of how the written code is run

in Python’s virtual machine. The debugger works by pausing execution of code upon reaching a statement. Debugging is started by pressing the “Debug current script” button or with “Control” + “F5” key combination, which automatically highlights the first statement. The user can then choose to step into or step over the statement. Executing either command automatically updates the variables table and shell output according to the program being run and statements evaluated.

If the developer uses the “Step into” command, for which the corresponding toolbar button or “F7” key can be pressed, the execution of the highlighted statement is visualized. The focus indicator moves to highlight the statement’s first child. When an expression is encountered, further stepping results in gradually evaluating the expression. A box appears, where a copy of the expression is initially displayed as seen on Figure 5. By choosing “Step into” again, the first child of the expression is selected. Then stepping again replaces the child with its value – for example, a variable’s name is replaced with its value or the sub-expression is evaluated. Note the evaluated sub-expression’s value “16” coloured in blue on Figure 5. When the statement’s children have been evaluated, the focus goes back to the parent statement and stepping again moves the highlighter to the next statement.



The image shows a code editor window titled 'calculate.py'. The code contains the following lines:
`a = 4`
`b = 2`
`rms = sqrt((16 + b**2) / 2)`
`print(rms)`
The expression `sqrt((16 + b**2) / 2)` is highlighted with a light blue background. Within this expression, the sub-expression `16 + b**2` is highlighted with a yellow background. The value `16` is displayed in blue text, indicating it has been evaluated. The variable `b**2` is highlighted with a yellow background, indicating it is the next sub-expression to be evaluated.

Figure 5. Thonny’s debugger displaying the evaluation of a root mean square expression.

Another option for moving forward in the debugging mode is to choose the “Step over” command by clicking the accordingly named button or pressing “F6” on the keyboard. This command instantly executes the current statement and moves the pointer to the next statement or evaluates the highlighted expression by replacing it with its value.

Thonny does not feature setting breakpoints. Instead, “Run to cursor” can be used. This command works by first setting the cursor’s location in the editor to the requested spot and

then issuing the command by using the key combination “Control” + “F8” or by choosing the command from the “Run” dropdown menu. Then, the debugger executes the source code up until to the selected statement in the source code.

A further command available to users is “Step out”. Issuing this command instantly completes the code currently in focus and all following sections of code that are on the same level – for example, if the evaluation of function arguments is in progress, all the arguments of the current function are evaluated and the focus returns to the function call. When issuing this while executing a function, the function will be executed and the pointer stops at the function’s return value. When issuing the command while evaluating a child of a statement, the child is instantly evaluated and the next statement will be selected.

Another prominent feature of Thonny’s debugger is its visualization of function calls. After the final argument of the function call is evaluated, the whole call expression is highlighted. If “Step into” is chosen thereafter, a small window displaying the function’s body is opened. The local variables of the function are also displayed, illustrating the concept of scopes. The developer can then proceed stepping as before. Additional function calls open additional windows, as displayed on Figure 6. After completing the function’s execution, the window is destroyed and the previously highlighted function call is replaced with its return value.

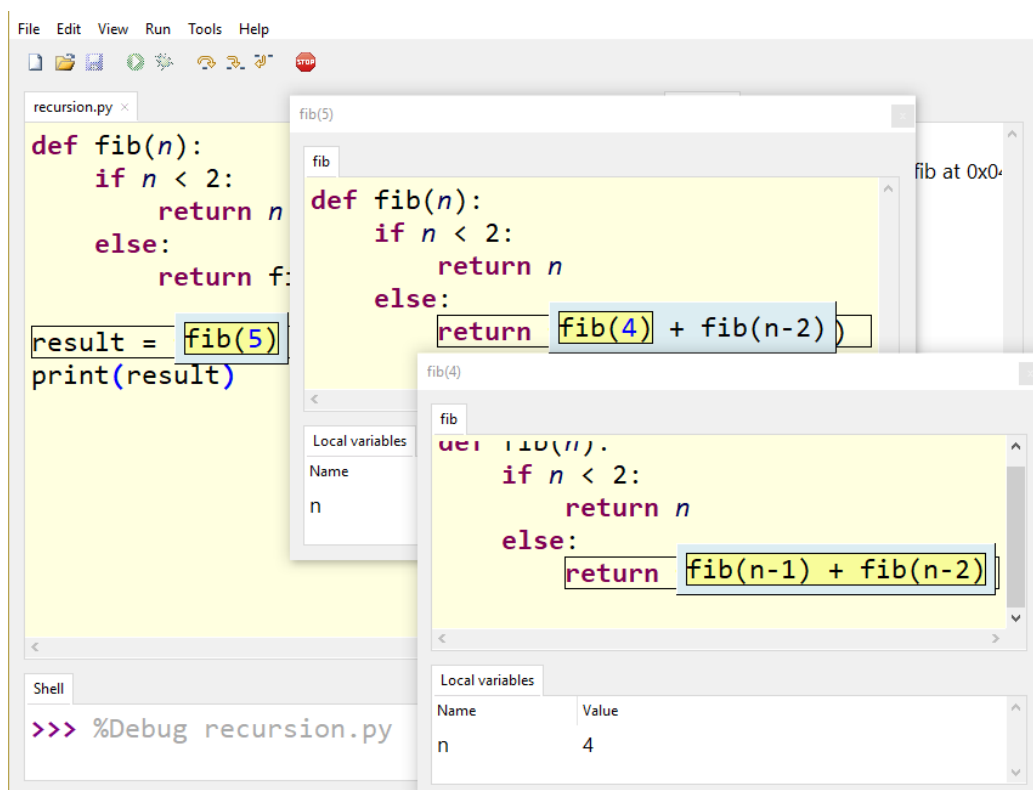


Figure 6. Debugging a recursive function in Thonny.

Thonny's features are all aimed at the novice developer studying in introductory programming courses. Featuring many different views and visual aids for learning basic concepts, using Thonny can help beginners learn programming faster than coding with advanced IDEs. Being open source and free to use, it also allows users to develop their own versions of Thonny.

3 Documenting the Back-end of Thonny

In addition to being a Python 3 IDE, Thonny is itself written in Python 3. Thonny uses two processes at runtime – first of them is for the graphical user interface (GUI) front-end which is based on Python’s TkInter [17] framework, the second is for the back-end handling the execution of user code. Before the development of any feature for Thonny, the back-end’s structure must be understood. The following section (accurate as of Thonny version 2.1.16) gives an overview of the architecture of Thonny’s back-end.

3.1 Main Logic

Thonny supports all Python 3 programs. To facilitate this, Thonny’s back-end makes use of Python’s tracing tools. To support statement-based stepping and gradual expression evaluation provided in Thonny’s debug mode, additional information must be provided to the tracer to better determine the current location in code. This is done by adding marker nodes (special function calls created by Thonny) to the program’s syntax tree before compilation and execution. For each statement, markers called *BEFORE_STATEMENT* and *AFTER_STATEMENT* are added just before and after the original statement respectively. Those functions contain additional data about the program’s current state: where the statement is in the code, does the statement have children etc. Likewise, each expression is surrounded with marker expressions *BEFORE_EXPRESSION* and *AFTER_EXPRESSION*, signalling the tracer that an expression evaluation is about to begin and providing the value of the evaluated expression afterwards [4].

The routine of Thonny’s back-end, illustrated on Figure 7, starts with the main loop, which first waits for a command from the front-end. If a command is received, it is read and passed on to the command handler, which chooses the suitable executor for the command. For running the program, *Executor* class will be chosen. Just as the name implies, this class is meant for executing the developer’s code. For debugging the script, *FancyTracer* class will be selected. This class is an advanced representation of Python’s tracing capabilities, making use of the additional information needed for statement and expression based debugging of Thonny. After the handler has finished, the result is passed to the front-end and the main loop resumes. In case the developer chooses to interrupt execution by clicking the “Interrupt/Reset” button or by using the “Control” + “F2” key combination, the currently running command is stopped and a corresponding notice is sent to the front-end instead.

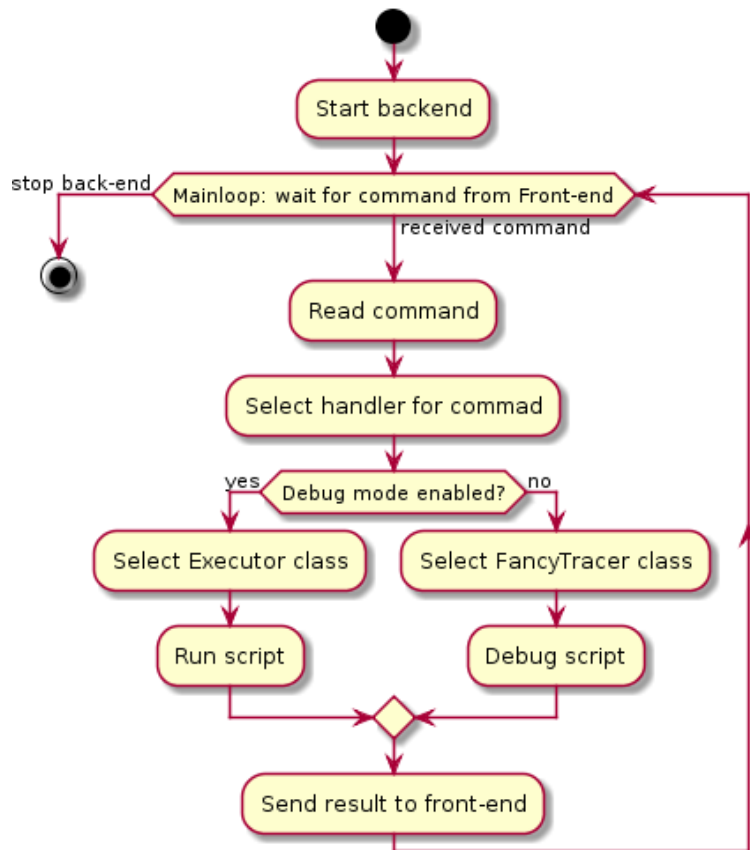


Figure 7. Diagram illustrating the main loop of Thonny’s back-end.

In addition to the main command-handling routine described here, the developer must also know the concept of the script debugging sub-routine before any upgrade to the debugger can be implemented.

3.2 Debugging Logic

If the user chooses to debug the current script, the debugging subroutine is initiated, which is also described by Figure 8. First, markers are placed in the executable code. Then a “Step” command is automatically issued, which signals the back-end to report the state upon reaching the first *BEFORE_STATEMENT* marker. Then Python’s standard library’s *exec* function is called, which signals the Python virtual machine (VM) to start executing code with tracing enabled. As debugging is initiated, a piece of code is executed and then the *_trace* function is called by the VM. This is when Thonny’s custom tracer implemented in the *FancyTracer* class starts, calling the *handle_progress_event* function in the process. This function determines whether the observed state should be reported to the front-end and if so, asks for the next command. The next command then sets the conditions for the next state upon which the current state should be reported to the front-end. When the *_trace* function

returns, the VM executes the next piece of code and calls the `_trace` function again. This is repeated until the program execution concludes. Notice that the parts handled by Thonny’s logic are in the boxes named “Thonny” on Figure 8. Unboxed areas like code execution and `_trace` function calls are handled by Python’s VM.

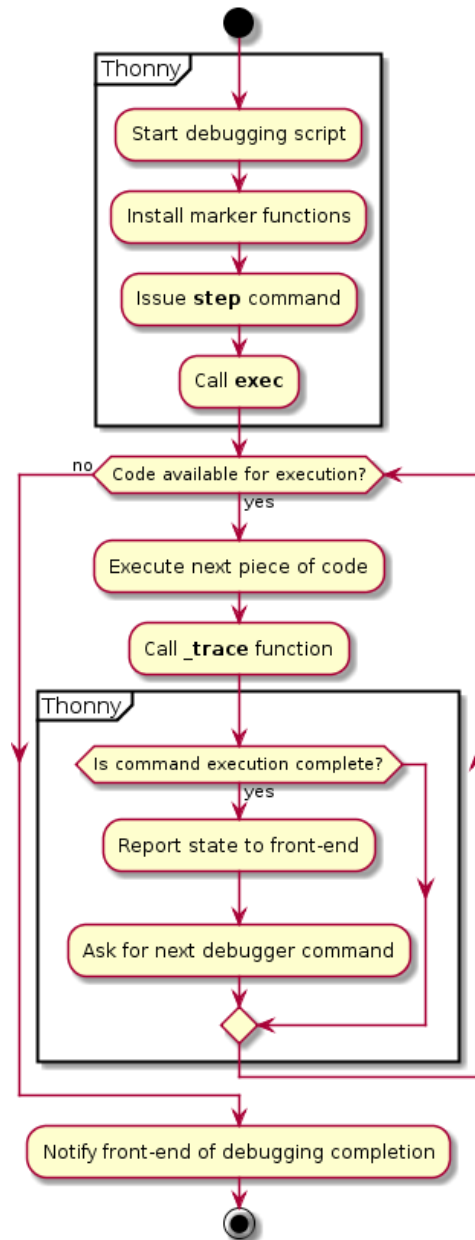


Figure 8. Debugging mode routine. Parts handled by Thonny are in corresponding boxes.

Upon receiving debugging commands, the tester function, which determines the correct state to report to the front-end, checks if the VM’s code execution has reached a state, where the current command is complete. For example, if the user calls the “Step into” command, the next state is displayed to the user. If the user chooses “Step over” instead, the VM runs to the next *AFTER* marker, which is at the same level as the state where the command was

issued. For example, if “Step over” is issued on a *BEFORE_EXPRESSION* marker, the state on next *AFTER_EXPRESSION* marker will be reported. When issuing “Step out”, execution proceeds to the next state, where the frame, on which the command was issued, has completed. Lastly, for “Run to cursor”, the execution is concluded when the *BEFORE_STATEMENT* marker of a statement is reached, on which the debugger’s focus matches the cursor’s location. Once the debugger has determined that the command’s execution has finished, the current state is reported to the front-end and the next command is polled, after which the VM executes the next piece of code and calls the *_trace* function again. Upon reaching the end of the running script, the last state is reported and the debugging mode will exit.

On some states, the front-end may determine that the current execution has reached an *AFTER_STATEMENT* marker (for example, issuing “Step over” on *BEFORE_STATEMENT* could lead to this). On these occasions, the front-end automatically issues an internal debugger command called “Run to before”, which tells the back-end to report the next state only if the VM’s code execution has reached the next *BEFORE_STATEMENT* marker. This reduces the number of stepping commands needed from the user to move from statement to statement – skipping *AFTER_STATEMENT* states makes debugging smoother.

Although Thonny is meant for novice programmers, the versatility of its back-end makes using Thonny worthwhile even for experienced developers, as it can run any Python 3 program. The existing architecture makes it possible to introduce upgrades like developing the ability to step back in time. The provided description of the back-end could also help future contributors grasp the structure of the back-end, possibly hastening the development of further upgrades for Thonny.

4 Implementation of Omniscient Debugging

Considering available concepts for implementing reversible debuggers, it was decided that stepping back will be implemented based on the omniscient debugging method. The reason being that other reversible debugging approaches may necessitate an extensive redesign of Thonny’s back-end while omniscient debugging can be incorporated into existing debugging architectures. We also thought that true-reversible debugging may confuse novice developers more than replaying past states. As the changes will be directly written into the existing source code of Thonny’s back-end, it was decided that the solution will be written in Python 3.

The development of Thonny’s omniscient debugger was done in cooperation with Aivar Annamaa, the supervisor of this thesis and author of Thonny. The author of this thesis made changes primarily in the back-end of the IDE, while Annamaa made minor changes to the front-end to eliminate some assumptions made when the debugger could run forward only. Development was done in a forked repository of Thonny and changes were merged into Thonny’s main repository once no major bugs could be observed while debugging a program. The commits of the author of this thesis can be seen by referring to the commits page of Thonny’s main repository and searching for commits by Alar Leemet [18,19].

The core idea for implementing omniscient debugging for Thonny was to change *handle_progress_event* (the function for determining whether we should report the current program state) in a way that it does not return after receiving a command while the front-end is displaying a past state or a backwards stepping command is issued. The returning of that function would signal the Python virtual machine to resume executing code, which should be avoided unless new code needs to be executed. Additionally, an infrastructure for saving states and selecting the correct state from the saved messages is needed to access past states of a program. A new command must also be introduced, that is able to select a state from the past (which will be called “Step back”). The “Step back” command will be implemented as a “Undo” command that will restore the previously displayed state. This will make it easier to play and replay the execution of the program or return in front to the state that the user may have accidentally skipped. Finally, the logic of determining debugger command completion must be changed so that the right moment can be detected via previously saved data. Displaying past states based on recorded data is the main trait of omniscient debugging.

4.1 Initial Attempt

At the first stages of development, it was decided that an initial proof of concept will be developed, which should give an idea of what should be done for record-replay debugging to function properly in Thonny. It had to be able to save minimal amount of data needed to display past states and had to be able to handle stepping back by going through every previously saved state. Once this had been completed, more functionality was to be added, removing the shortcomings of the initial solution.

The first attempt was based on saving call stacks after every executed piece of code, as they contain most of the data needed to display past states. For this, a field named *_past_stacks* was added to the *FancyTracer* class, storing the saved call stacks in a list. The main logic was written directly into the *handle_progress_event* function so it would be launched only if a “Step back” command is received. The routine starts by setting a pointer, which keeps track of the currently selected stack in *_past_stacks*. Then, the routine takes over sending states and receiving commands, moving the pointer forwards or backwards across the saved states, depending on the command received. This attempt did not feature any logic that could determine the completion of a more complex command (like “Step over”) in the past. This meant that once stepping in the past was initiated, every forward-executing command like “Step”, “Step over” or “Step out” moved the pointer one step forward and “Step back” moved the pointer one step backward.

During the testing of this iteration, it was discovered that the assumptions made by the front end prevented the initial solution from functioning. The front-end’s automatically issued “Run to before” command told the back-end to report the next state when the “Step back” command was issued on a *BEFORE_STATEMENT* marker. The previous state should have been reported instead. Additionally, the initial solution did not properly prevent *handle_progress_event* method from returning while stepping through past states, resulting in some states not being displayed as they should. Furthermore, it was discovered that the data in the saved stacks is not enough for Thonny to reproduce some states in the front-end. This iteration can be found in Thonny’s repository [20].

4.2 Interim Solutions

The initial solution did not provide the front-end enough data to properly display expression evaluation. The back-end had to additionally send all the values of the currently evaluated

children of the expression. Otherwise the front-end would be unable to display values for sub-expressions while stepping through past states. This was also implemented, although it did not prove to be enough to solve this issue.

Later we found that it would be more convenient to save the required data directly as messages, providing all required details about given program state. It would give the ability to instantly report the state if requested. Thus, the field *_past_stacks* was replaced with *_past_messages*. For this, the initially implemented architecture had to be overhauled.

To support message saving, the *_report_state_and_fetch_next_message* function was split into two methods, which was in charge of sending states to front-end and receiving the next message. First of them is the *_save_debugger_progress_message* method, which handles message creation and saving these into *_past_messages* for past state browsing. The second method is *_send_and_fetch_message*, which sends the message to the front-end and polls the next message. The splitting of the function gives more flexibility – as each message can be sent multiple times, message saving may not always occur before sending a message. Additionally, all the messages are now saved regardless if the message would be sent to the front-end at all. Thanks to this, if the user accidentally steps over the most interesting part of code, the user can step back and step through the previously skipped part again using smaller steps. Also, stepping over code was now possible while browsing past states. This iteration also brought the main logic under a separate function because the *handle_progress_event* function grew larger with added functionality.

The extra features also added more complexity to the solution. The biggest problem was associated with logic repetition – the same approach was essentially used for both determining the place to respond to the debugger command and to signal to the browsing pointer that the selected message should be sent to the front-end. Also, the “Step over” command ceased to function under certain conditions, as the “Run to before” command was disabled while a past state was displayed. At the same time, “Run to before” couldn’t be enabled because it would then prevent stepping back as mentioned before. Additionally, some states still did not provide enough data for the front-end to display properly. This implementation can be found by referring to commit [47701e5eec88](#) in Thonny’s repository [21].

To remove the logic repetition, we agreed that the code for stepping forward in time and browsing past states should be merged and generified for both past and present use. For this, the core logic was merged into the *handle_browsing_past* function. The main idea was that

the code, that previously told when to report the current state to the front-end, should now move the pointer in the necessary direction by themselves and return the pointer's value when the appropriate state has been reached. Then the message chosen by the pointer was to be sent to the front-end. Distributing the moving of the pointer helped to counter issues with "Run to before" as seen in previous iterations of the solution.

Additional problems arose with this implementation. The biggest difficulty was to change the logic so that it could signal if the currently saved states are enough for the current command to complete. If the command could not complete, the function had to return to allow the VM to continue executing the program. When this was implemented, it turned out that the logic was not functioning in different corner cases. The "Step over" command still got stuck at *AFTER_STATEMENT* markers or did not stop at some *BEFORE_STATEMENT* markers in past states. This iteration can be found by referring to commit [27373a57d78b](#) in Thonny's repository [22].

4.3 Final Solution

As the changed approach caused many problems and moving pointers in the command completion sub-methods was a repetition of logic, a change was necessary again. Before the implementation of the eventual solution, some modifications to the front-end were also made. Among them was the removal of the "Run to before" command, as it caused many problems while stepping through past states. This also simplified the structure of the front-end. Finally, the missing data needed by the front-end to reconstruct past states was provided by saving the root expression and the completed evaluations of the state. The final set of data that must be saved for each state to properly display them on demand is the following:

1. Execution stack – holds frames containing information about the current statement and its location in code, the evaluated sub-expression values and function calls.
2. Exception data – information on current unhandled exception.
3. List of loaded modules – needed for updating the variables view.
4. Symbol counts for each stream (input, output, error) – to hide or show the symbols displayed in the shell according to the currently displayed state.
5. Is new – a flag for identifying if this is the latest state.

This is when the prerequisites for a functioning solution were established.

The idea of the final solution is similar to the debugging architecture in place before this thesis. When the previous solution gave the logic of checking command completion the ability to alter the state pointer, the final solution’s approach only had to tell if the pointer was currently at the right spot for responding to the command. This means that moving the pointer was centralized, reducing code repetition and simplifying the algorithm. Additionally, the logic for determining the completion of “Step over” was simplified: if the command is issued on a **BEFORE** marker, the command is complete when the focus has moved from a child to a parent or to another line; if the command was issued on an **AFTER** marker, the command is completed whenever the focus or marker changes. The change was necessary as the **AFTER_STATEMENT** markers are no longer skipped by “Run to before”. As of now, command completion is never signalled on **AFTER_STATEMENT** markers in order to make the debugging experience smoother. The majority changes can be seen in the commit [7521ae1a819](#) of Thonny’s repository and some minor modifications can be found in later commits [23].

The debugging routine, which structure can be seen on Figure 9, begins with the same steps as before the implementation: marker functions are installed; the initial step command is issued and the Python VM starts code execution in trace mode. Each time a piece of code is executed, the current program state is saved and then the *handle_message_selection* function is called, checking if the pointer is at a saved state. If so, the logic checks if the pointed state meets the current command’s completion requirements. In case the requirements are met, the pointed state is reported and the back-end will wait for the next command. After receiving the next command, the pointer is moved in the direction of that command – backward for “Step back”, forward otherwise – and the pointer’s current location is again checked. If the current state should not be reported to front-end, the message sending part is skipped. Once the pointer is not at a saved state, the *handle_message_selection* function returns, permitting the VM to continue executing the next piece of code. This routine is repeated until code execution finishes.

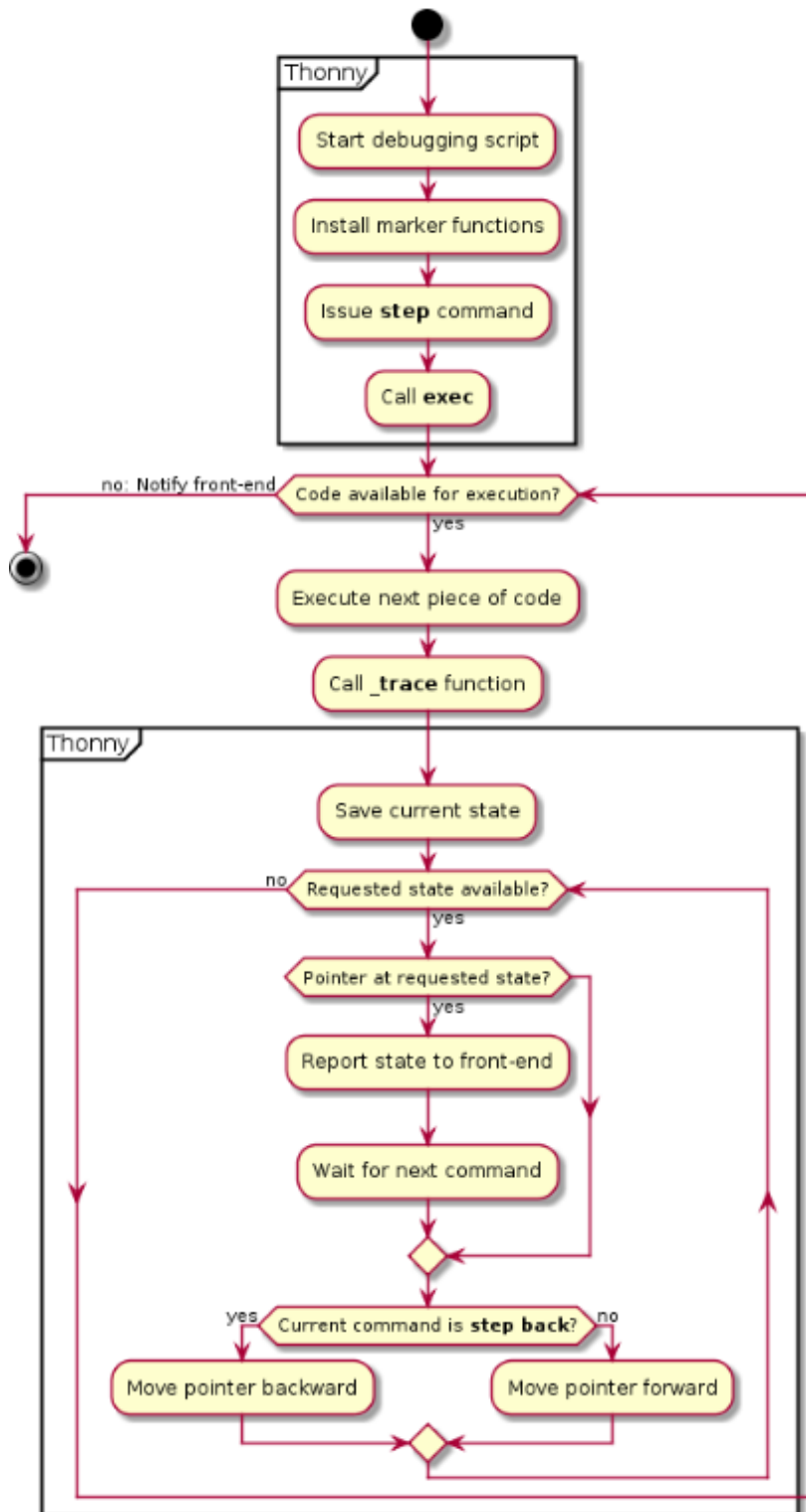


Figure 9. The final solution's logic. Parts handled by Thonny are in corresponding boxes.

The final solution proved to solve issues that plagued the interim solutions and simplified some logic for determining the correct state to report. It must be stressed though that this implementation is not true-reversible debugging, as stepping back and forward again does

not result in the line of code being re-executed, but the previously saved state being displayed again. This means for example, if a random number generation statement is stepped over, no new number is generated, but the previously generated number is again displayed. This may make things easier for beginners, but it must be made clear to them to avoid mixing up replaying execution and re-executing code.

Thanks to the existing universal architecture documented in the previous section, the implementation of an omniscient debugger could be done by making minor changes to the front-end and incorporating the main solution into the existing functions. Doing so, some parts of Thonny's front- and back-end were simplified. The added functionality should help beginners debug their programs, giving them the chance to go back and forth in saved states, comparing them and making sure that their program works as expected.

5 Testing of Thonny’s Omniscient Debugger

Testing of the omniscient debugging capabilities are carried out as part of Thonny version 2.2 beta program. For this, Thonny version 2.2.0b2 was released on 04.05.2018, featuring record-replay debugging among other new functionalities. The beta program will assess the feasibility of the implemented features with the help of the voluntary Thonny user community and their feedback. Based on the feedback, decisions will be made whether to keep and/or change the introduced features. The beta program was announced in Thonny’s blog [24] and supporting social media pages and forums.

5.1 Opinion Poll on Thonny’s Debugger

To gather feedback specific to the omniscient debugger, a Google Form was created. The form polls for the user’s opinions on the usefulness of the debugger, previous experience on omniscient/reversible debuggers and their comparison to Thonny’s debugger and user experience on how the current implementation performed. The poll is available in the blog post under the “Stepping back in time” section [24] and as a direct link access [25].

Initial poll feedback suggested that the debugger’s omniscience should be made an optional feature, that can be turned on or off from the options menu. The reasoning was that saving previous states has too much overhead, causing the program to use gigabytes of memory instead of megabytes (as it was before the new feature) and leads to eventual memory errors. The reporter submitted a code example, consisting of a simple for-cycle generating random numbers for 10000 iterations. Despite the issues, the answer also pointed out the usefulness of browsing past states for people learning programming – the main motivation for implementing the feature for Thonny.

5.2 Testing Memory Usage of Thonny

Regarding overhead issues reported in the poll, a test on Thonny’s memory usage was conducted to compare memory consumption before and after implementing omniscient debugging. Testing was done with 7 different programs, all of which are solutions to various programming exercises as part of University of Tartu’s Computer Science 1 course. The solutions were written by a freshman informatics student, who started learning programming as part of the same course. Beginner level programs were chosen for testing because these may match the programs that Thonny’s target audience may write.

Debugging the test programs with Thonny was monitored with Python’s *memory_profiler* module [26]. The module supports recording Python application’s memory usage throughout the lifetime of the application. Each test session consisted of launching Thonny, starting the debugging of the program, issuing the “Step out” command (which completes the program execution as one single step) and exiting Thonny after the debugging of the program has finished. The results of each test were visualized by *memory_profiler*’s graph plotter. The test programs, exercise texts and the memory graphs containing test results for each exercise and both versions of Thonny can be found in the appendix.

As seen on Figure 10, results indicate that the omniscient debugging feature increases the memory consumption of Thonny’s back-end. Without the new feature, Thonny’s back-end uses on average a maximum of about 12.71 MiB of memory during its lifetime, regardless of the program being executed. Testing results with omniscient debugging show great fluctuations of memory usage between different programs – peak memory usages range from about 21.91 MiB for the Tic-Tac-Toe exercise to about 421 MiB for the Nearest Points task. On average, the memory usage for Thonny’s back-end for the tested programs increased by about 7.38 times, to about 93.8 MiB. It is safe to say that this memory usage is not too high for contemporary computing standards, but as Thonny’s front-end used about 35 MiB of memory during all tests and combined with the back-end’s 13 MiB, 129 MiB is still a lot more memory used than 48 MiB.

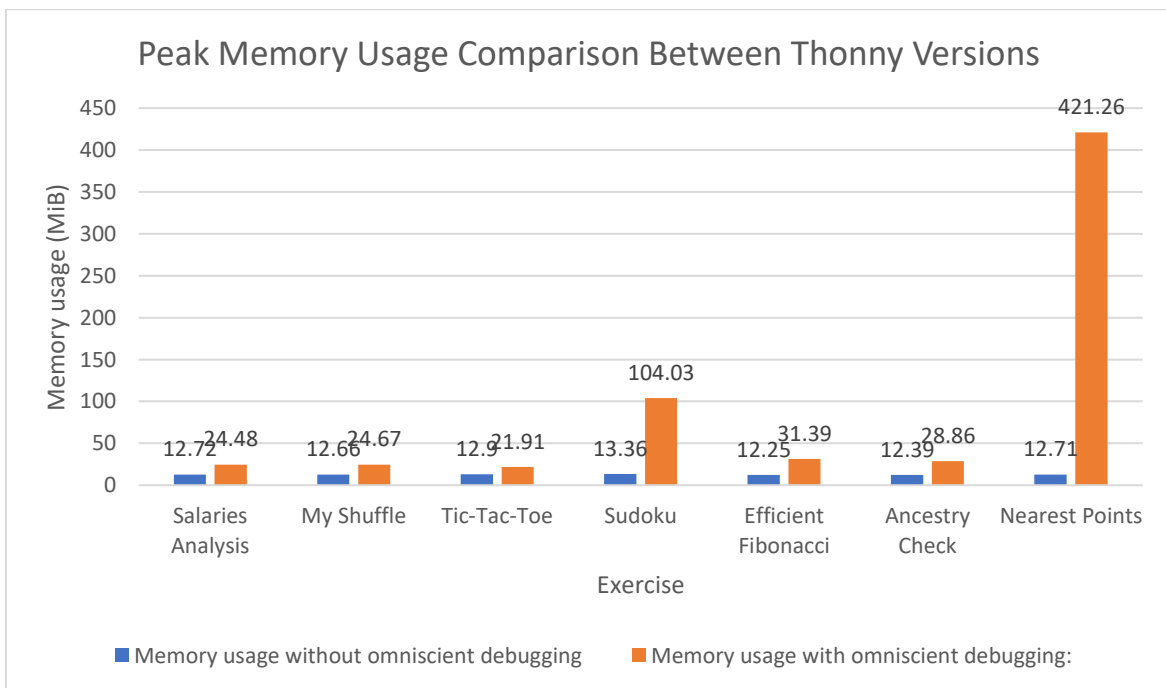


Figure 10. Peak memory usage of Thonny’s back-end with and without omniscience.

It must also be noted that the total debugging time noticeably increased on some occasions. While there were tasks where the time remained identical regardless of the implementation tested (like Salaries Analysis and Efficient Fibonacci tasks), there are cases where the debugging time more than doubled. A pattern has emerged – the more steps are needed to complete the execution of the program, the longer is the debugging session and the more memory Thonny consumes. For example, when the memory usage increased from 13 to 421 MiB on the Nearest Points task, the debugging time also increased from 10 seconds to 105 seconds, which means that the user had to wait about 10 times longer for the debugger to complete execution. But the difference in debugging time is not noticeable when the user chooses to step through the program with small steps, as the user’s reaction time is far greater than the process of executing code and saving states.

While the current implementation of omniscient debugging for Thonny may cause performance and usability issues with some programs, there are still cases where the difference in execution time and memory usage is not so significant. This what literature analysis also suggested. Thus, the feature should be made optional, as it may make debugging simple programs easier for beginners, but can be disabled once the user is more confident in his/her skills. This feature will be added before beta testing concludes. As the beta program is still in progress, any further deficiencies will be addressed by the author as soon as they are discovered.

6 Conclusions

This thesis provided an implementation of omniscient debugging capabilities for Thonny, a Python 3 integrated development environment for novice developers. Additionally, a literature study on debugging techniques was conducted, introducing different conventional and reversible debugging methods. Also, an overview of Thonny and its features was provided, describing the functionalities making learning programming easier. The back-end of Thonny was documented as well.

The new debugging options integrated for Thonny's debugger made it possible for users to browse past program states saved by the debugger. When the developer now accidentally steps over the most interesting part of code, the command can be undone and the same piece of code can be stepped through with smaller steps without restarting the debugging session. As all the states are saved, all the issued commands can be undone and the code can be stepped through any way the developer wants. Different concepts of implementing omniscient debugging into Thonny's existing debugger were tested, the final of which was included in Thonny version 2.2. The changes are being evaluated as part of Thonny's beta program, which is still in progress at the submittal of this thesis.

Initial testing results suggested that the solution should be made optional as is. Although omniscient debugging helps novice developers learn programming, high overhead caused by saving previous states can trigger memory errors or increase debugging time, significantly decreasing Thonny's usability while debugging certain programs. This was confirmed by additional memory profiling done with the versions of Thonny that have omniscient debugging and not. Omniscient debugging will be made optional until a more efficient solution for saving states is implemented, which can be done as further work on this subject. An option for improvement is to only save differences between program states (also called deltas) and reconstruct requested states based on the saved deltas. Any other shortcomings will be addressed as soon as they arise during the beta testing, after which the final decision on whether to keep omniscient debugging part of Thonny will be made.

7 References

- [1] Rouse M, Silverthorne V, Heusser M, Davis T. What is debugging? - Definition from WhatIs.com. <http://searchsoftwarequality.techtarget.com/definition/debugging> (04.03.2018).
- [2] Hailpern B, Santhanam P. Software debugging, testing, and verification. *IBM Systems Journal*, 2002, No. 41, pp 4-12.
- [3] List of debuggers - Wikipedia. https://en.wikipedia.org/wiki/List_of_debuggers (05.03.2018).
- [4] Annamaa A. Introducing Thonny, a Python IDE for learning programming. *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. Koli, 2015, 117-121.
- [5] Brady F. PRWeb. <http://www.prweb.com/releases/2013/1/prweb10298185.htm> (02.04.2018).
- [6] (NIST) NIOsaT. Software Errors Cost U.S. Economy \$59.5 Billion Annually: NIST Assesses Technical Needs of Industry to Improve Software-Testing. http://www.abeacha.com/NIST_press_release_bugs_cost.htm (13.05.2018).
- [7] Sabin P. Implementing a Reversible Debugger for Python. Vienna University of Technology, Faculty of Informatics, Master's Thesis. 2010. <https://liinwww.ira.uka.de/cgi-bin/bibshow?e=Dpnqjmf0Jotu/Dpnqvufstqsbdif0/UVXjfo/fyqboefe%7d37:158&r=bibtex&mode=intra>.
- [8] Pothier G. Towards Practical Omniscient Debugging. University of Chile, Faculty of Physical and Mathematical Sciences, PhD Thesis. 2011. <http://repositorio.uchile.cl/handle/2250/102687>.
- [9] Engblom J. A Review of Reverse Debugging. *Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D)*. Vienna, 2012, 1-6.
- [10] Boothe B. Efficient algorithms for bidirectional debugging. *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. Vancouver, 2000, 299-310.

- [11] Coetzee AL. Combining reverse debugging and live programming towards visual thinking in computer programming. Stellenbosch University, Department of Mathematical Sciences, Master's Thesis. 2015.
<http://scholar.sun.ac.za/handle/10019.1/96853>.
- [12] Ilp T. Improving the Usability of the Thonny Integrated Development Environment. University of Tartu, Institute of Computer Science, Bachelor's Thesis. 2015.
http://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=46151&year=0.
- [13] Annamaa A. Thonny, a Python IDE for Learning Programming. *Proceedings of the ITICSE '15 Innovation and Technology in Computer Science Education Conference*. Vilnius, 2015, 343.
- [14] Thonny, Python IDE for beginners. <http://thonny.org/> (24.04.2018).
- [15] Long S. A Raspbian Desktop Update with some new Programming Tools - Raspberry Pi. <https://www.raspberrypi.org/blog/a-raspbian-desktop-update-with-some-new-programming-tools/> (04.03.2018).
- [16] plas / thonny - Bitbucket. <https://bitbucket.org/plas/thonny/overview> (05.03.2018).
- [17] TkInter - Python Wiki. <https://wiki.python.org/moin/TkInter> (01.05.2018).
- [18] plas / thonny / source / thonny / backend.py — Bitbucket.
<https://bitbucket.org/plas/thonny/history-node/f145b739b0cb/thonny/backend.py?at=master> (14.05.2018).
- [19] plas / thonny / source / thonny / plugins / debugger.py — Bitbucket.
<https://bitbucket.org/plas/thonny/history-node/4b5f35c31f02/thonny/plugins/debugger.py?at=master> (12.05.2018).
- [20] plas / thonny / commit / 0ec1001 - Bitbucket.
<https://bitbucket.org/plas/thonny/commits/0ec1001af579eae51dde58485744dbf9db0f8774> (04.05.2018).
- [21] plas / thonny / commit / 47701e5eec88 — Bitbucket.
<https://bitbucket.org/plas/thonny/commits/47701e5eec88e8ef5c657b0b0a1f8c713e4e440b> (05.05.2018).
- [22] plas / thonny / commit / 27373a57d78b — Bitbucket.
<https://bitbucket.org/plas/thonny/commits/27373a57d78bbb77f6e742037394bc96db2e1440> (05.05.2018).

- [23] plas / thonny / commit / 7521aea1a819 — Bitbucket.
<https://bitbucket.org/plas/thonny/commits/7521aea1a8192a0104e36fe26ea9d9b54d10495c> (12.05.2018).
- [24] Version 2.2.0 beta — Thonny blog.
http://thonny.org/blog/2018/05/04/version_2_2_0_beta.html (06.05.2018).
- [25] Opinion poll on Thonny's record-replay debugger.
<https://docs.google.com/forms/d/e/1FAIpQLSfb8anjFLRwZgZlq9juBEilmKhrjLhuJJZTnFT6L0gaKmBhIQ/viewform> (06.05.2018).
- [26] Pedregosa F, Gervais P. memory_profiler · PyPI.
https://pypi.org/project/memory_profiler (10.05.2018).

Appendix

I. Testing Resources

In this section, the programs used in testing and the corresponding testing results are presented. The format is as following:

1. A short description of the exercise.
2. The exercise text for which the program was made (or a link to the text).
3. The source code of the program.
4. The memory usage graph of the pre-omniscient debugging version of Thonny.
5. The memory usage graph of the omniscient debugging version of Thonny.

The graphs display memory usage of Thonny's front-end (black) and back-end (blue) separately. Notice the drop in the beginning of the back-end's graphs on each Figure indicate the starting of the debugging session. A total of seven programs were tested, which were all written by a freshman as part of University of Tartu's Computer Science 1 course.

1. Salaries analysis

The idea is to conduct an analysis of the people's salaries in the provided file.

Exercise no 2 at http://progeopik.cs.ut.ee/09_muteerimine.html#ulesanded.

```
f = open("palgad.txt")
andmed = f.readlines()
f.close()

nimed = []
vanused = []
palgad = []
for info in andmed:
    info = info.split(";")
    nimed += [info[0]]
    vanused += [int(info[1])]
    palgad += [int(info[2])]

suurim = max(palgad)
s = palgad.index(suurim)
keskmine = round(sum(palgad)/len(palgad))

k = 0
for palk in palgad:
    if palk > keskmine:
```

```

        k +=1

vanused_väiksem = []
m = 0
for palk in palgad:
    if keskmine >= palk:
        vanused_väiksem += [vanused[m]]
        m+=1
vanus_keskmine_väiksem = round(sum(vanused_väiksem)/len(vanused_väiksem))

vanused_suurem = []
n=0
for palk in palgad:
    if keskmine < palk:
        vanused_suurem += [vanused[n]]
        n+=1
vanus_keskmine_suurem = round(sum(vanused_suurem)/len(vanused_suurem))

print("Suurima palgaga töötaja on", nimed[s] + ", tema palk on", palgad[s],
"€.")
print("Keskmine töötajate palk on", keskmine, "€.")
print("Keskmisest palgast rohkem teenivad", k, "töötajat.")
print("Keskmisest palgast rohkem teenivate töötajate keskmine vanus on",
round(vanus_keskmine_suurem), "aastat.")
print("Keskmisest palgast vähem teenivate töötajate keskmine vanus on", va-
nus_keskmine_väiksem, "aastat.")

```

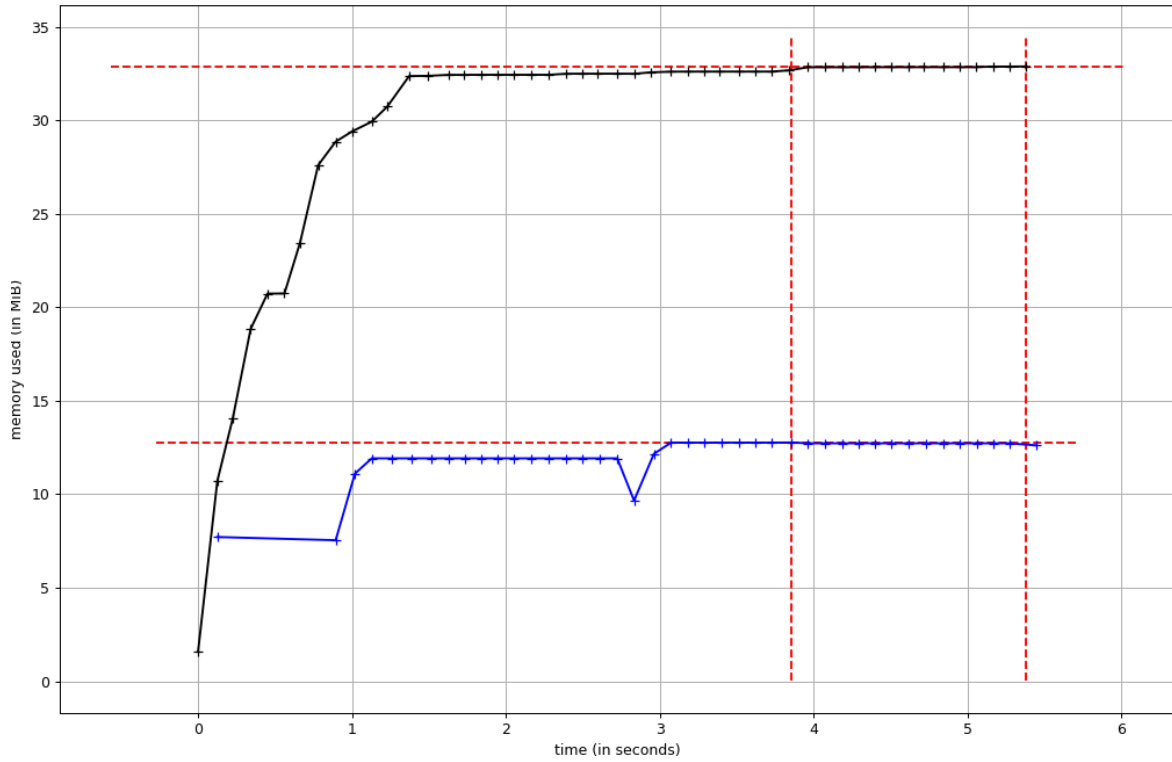


Figure 11. Memory usage by pre-omniscient debugging Thonny – Salaries analysis.

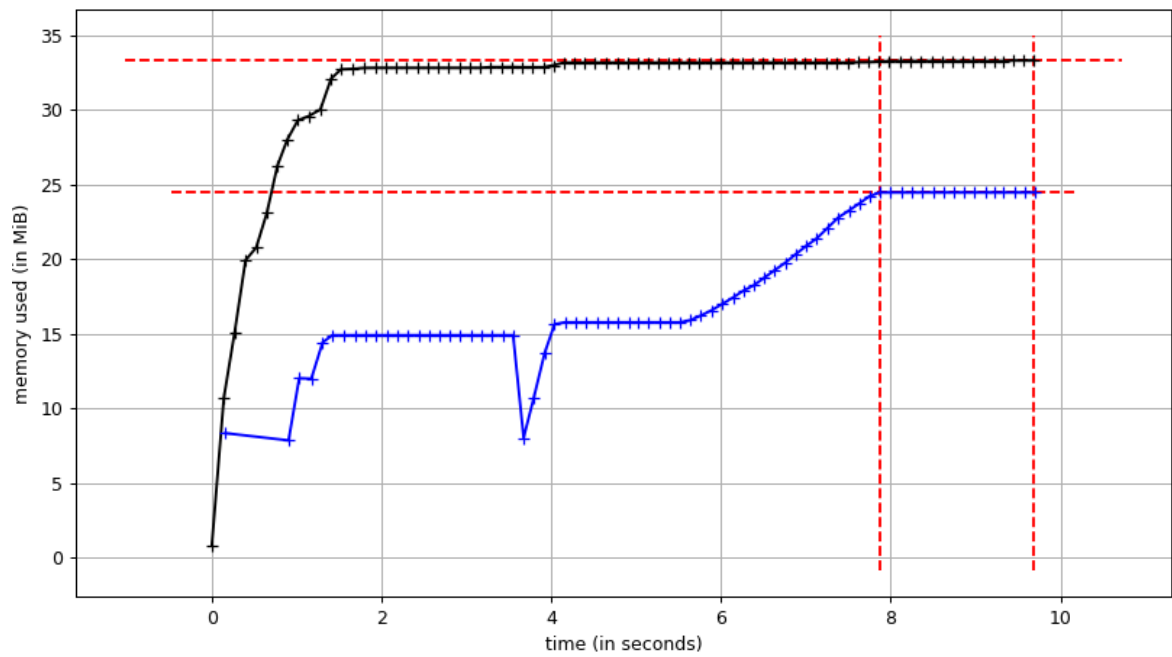


Figure 12. Memory usage by omniscient debugging Thonny – Salaries analysis.

2. My Shuffle

The programmer is asked to create his own version of a list shuffling function.

Exercise no 3 at http://progeopik.cs.ut.ee/09_muteerimine.html#ulesanded.

```
from random import *
def minu_shuffle(järjend):
    väljund = []
    pikkus = len(järjend)
    while pikkus != 0:
        uus = järjend.pop(randint(0, pikkus-1))
        väljund += [uus]
        pikkus = len(järjend)
    return väljund

a = [-2,-1, 0, 1, 2, 3, 4, 5]
print(minu_shuffle(a))
```

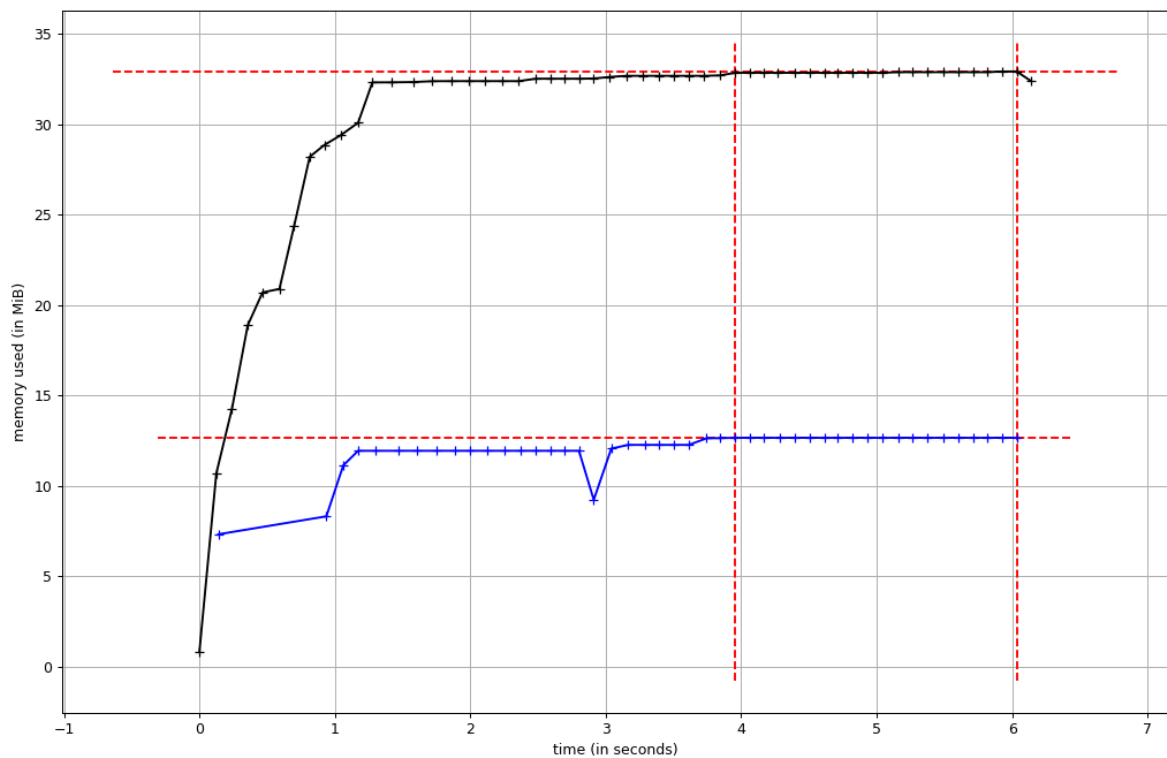


Figure 13. Memory usage by pre-omniscient debugging Thonny – My Shuffle.

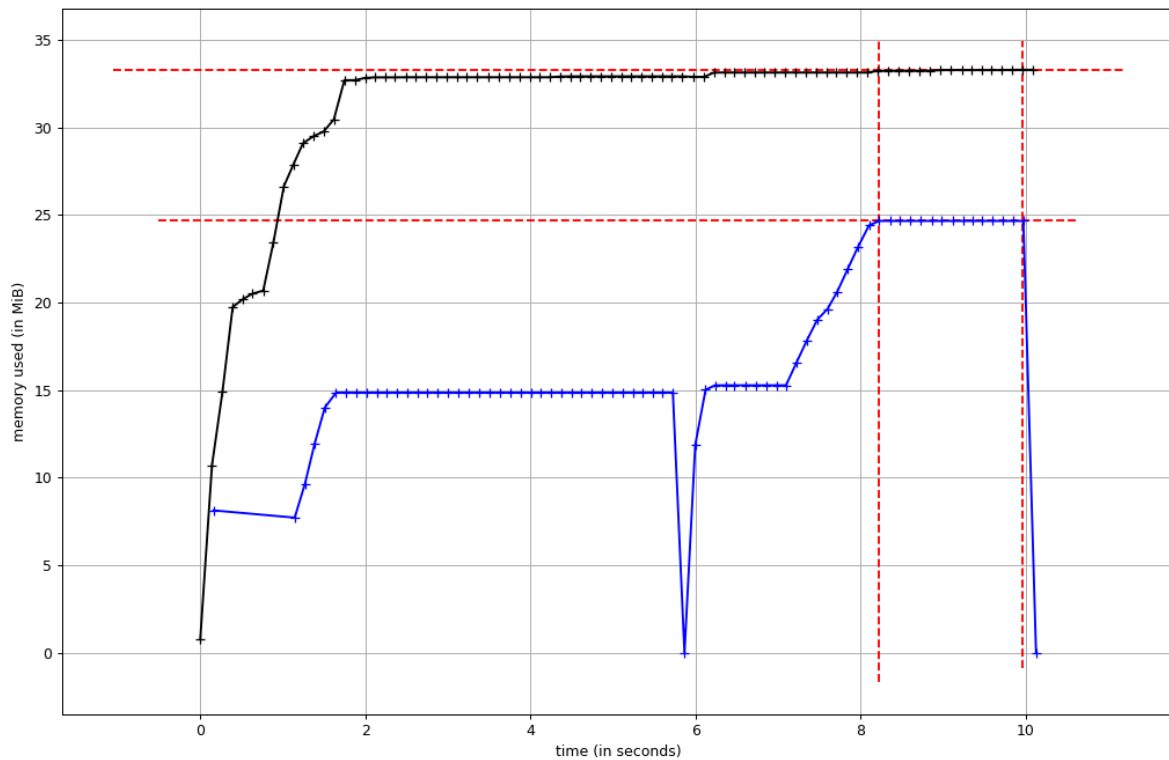


Figure 14. Memory usage by omniscient debugging Thonny – My Shuffle.

3. Tic-Tac-Toe

A function for determining the winner of a Tic-Tac-Toe game is implemented.

Exercise no 4 at http://progeopik.cs.ut.ee/10_andmestruktuurid.html#ulesanded

```
def võitja(mäng):
    # Kui on kasutatud muud sümbolikat kui X, O või " ".
    for rida in mäng:
        for veerg in rida:
            if veerg != "O" and veerg != "X" and veerg != " ":
                return "?"

    # Kui real on kolm järjestikust.
    võit = []
    for rida in mäng:
        if rida[0] == rida[1] == rida[1] == rida[2] and rida[0] != " ":
            võit += [rida[0]]

    # Kui selgub, et kolme järjestikust on rohkem kui üks.
    if len(võit) > 1:
        return "?"
    elif len(võit) == 1:
        return võit[0]

    # Kui veerul on kolm järjestikust.
    for k in range(0,3):
```

```

    if mäng[0][k] == mäng[1][k] == mäng[2][k] and mäng[0][k] != " ":
        võit += [mäng[0][k]]
# Kui selgub, et kolme järjestikust on rohkem kui üks.
if len(võit) > 1:
    return "?"
elif len(võit) == 1:
    return võit[0]
# Kui diagonaalil loodest kagusse on kolm järjestikust.
if mäng[0][0] == mäng[1][1] == mäng[2][2] and rida[0] != " ":
    return mäng[0][0]
# Kui diagonaalil kirdest edelasse on kolm järjestikust.
if mäng[0][2] == mäng[1][1] == mäng[2][0] and rida[0] != " ":
    return mäng[0][2]

return "?"

print(võitja([[ 'O', ' ', 'X' ],
              [ 'O', ' ', ' ' ],
              [ 'X', ' ', ' ' ]]))

```

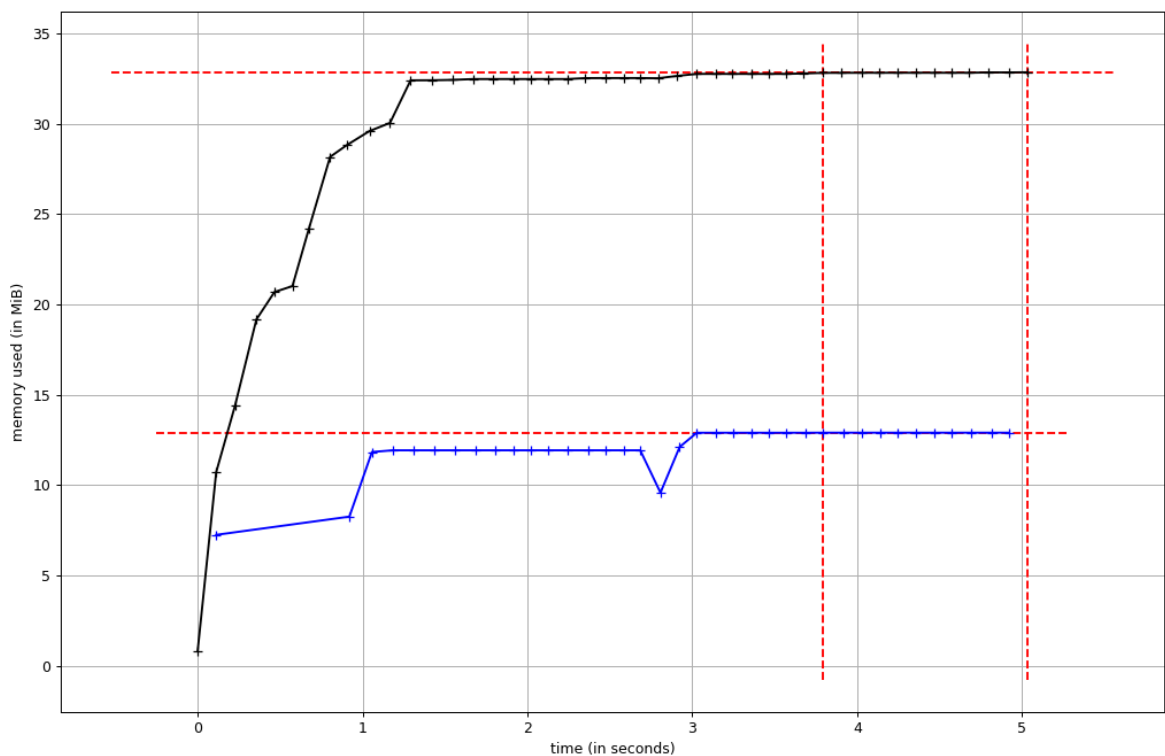


Figure 15. Memory usage by pre-omniscient debugging Thonny – Tic-Tac-Toe.

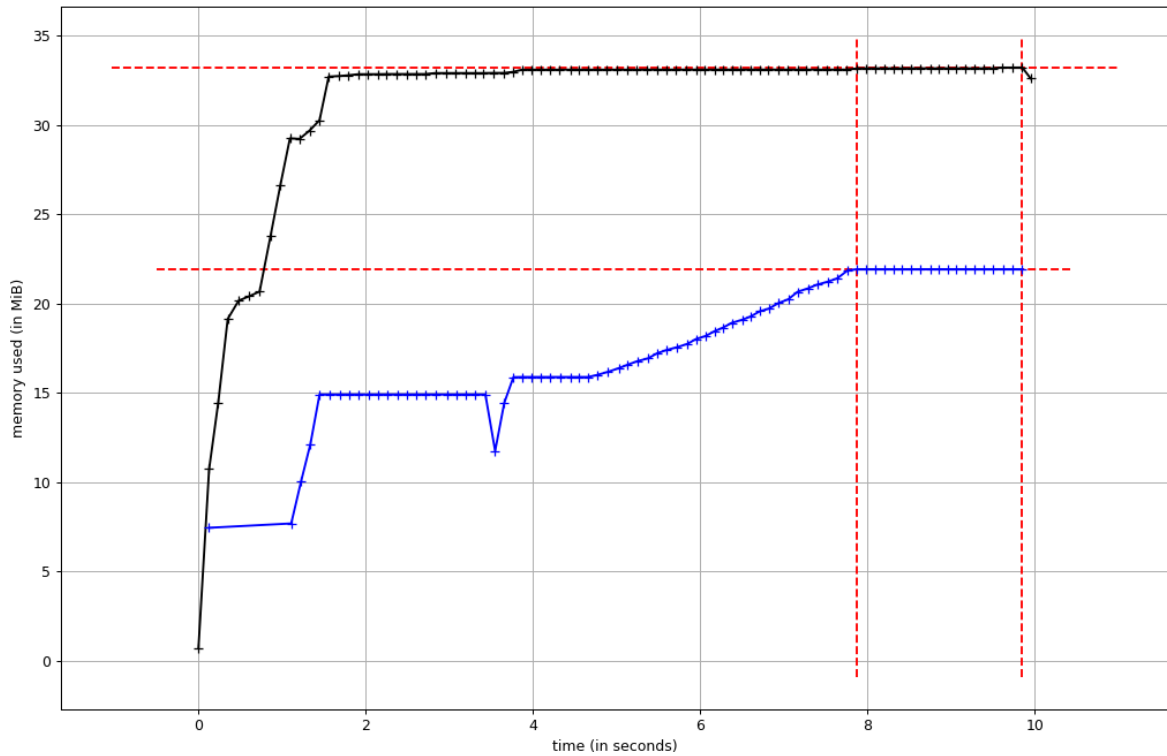


Figure 16. Memory usage by omniscient debugging Thonny – Tic-Tac-Toe.

4. Sudoku

A sudoku solution correctness checker is written.

Exercise no 5 at http://progeopik.cs.ut.ee/10_andmestruktuurid.html#ulesanded

```
import sys

def kontrolli_ridu(read):
    for rida in read:
        for i in range(1, 10):
            if str(i) not in rida:
                global viga
                viga = True
                print( "Viga "+ str(read.index(rida)+1)+". "+ "reas.")

def moodusta_veerud(read, veerud):
    for rida in read:
        i = 0
        for number in rida:
            veerud[i] += number
            i+=1

def kontrolli_veerge(veerud):
    for veerg in veerud:
```

```

        for l in range(1,10):
            if str(l) not in veerg:
                global viga
                viga = True
                print( "Viga "+ str(veerud.index(veerg)+1)+" ". "+ "veerus.")

def moodusta_ruudud(read, ruudud):
    j=0
    for i in range(0,9):
        if i < 3:
            ruudud[i]+=read[0][j: j+3]
            ruudud[i]+=read[1][j: j+3]
            ruudud[i]+=read[2][j: j+3]
        elif 2<i<6:
            ruudud[i]+=read[3][j-9: j-6]
            ruudud[i]+=read[4][j-9: j-6]
            ruudud[i]+=read[5][j-9: j-6]
        elif i<9:
            ruudud[i]+=read[6][j-18: j-15]
            ruudud[i]+=read[7][j-18: j-15]
            ruudud[i]+=read[8][j-18: j-15]
        j+=3

def kontrolli_ruute(ruudud):
    for ruut in ruudud:
        for i in range(1,10):
            if str(i) not in ruut:
                global viga
                viga = True
                print("Viga "+ str(ruudud.index(ruut)+1)+" ". +"ruudus.")

fail = "sudoku.txt" #sys.argv[1]
f = open(str(fail))
rows = f.readlines()
f.close()
viga = False
for row in rows:
    rows[rows.index(row)] = row.strip().split()

columns = [[], [], [], [], [], [], [], [], [], []]
squares = [[], [], [], [], [], [], [], [], [], []]

moodusta_veerud(rows, columns)
moodusta_ruudud(rows, squares)

```

```
kontrolli_ridu(rows)
kontrolli_veerge(columns)
kontrolli_ruute(squares)

if viga == False:
    print("OK")
```

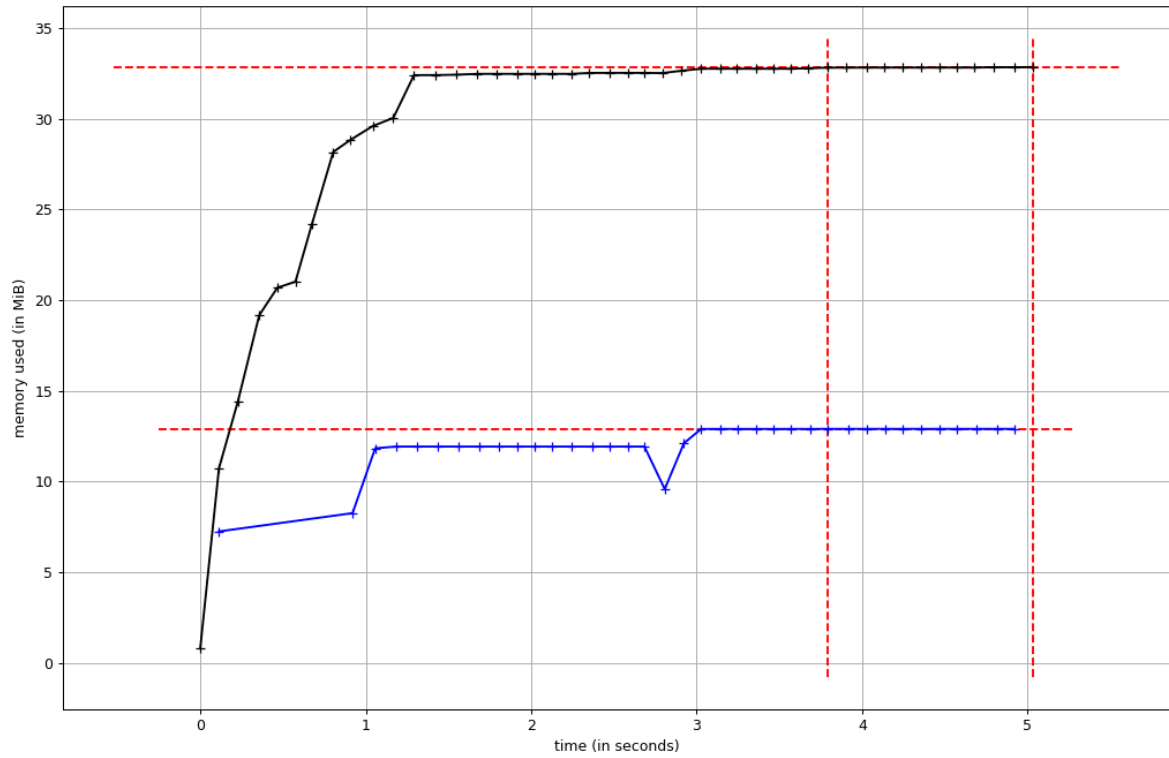


Figure 17. Memory usage by pre-omniscient debugging Thonny – Sudoku.

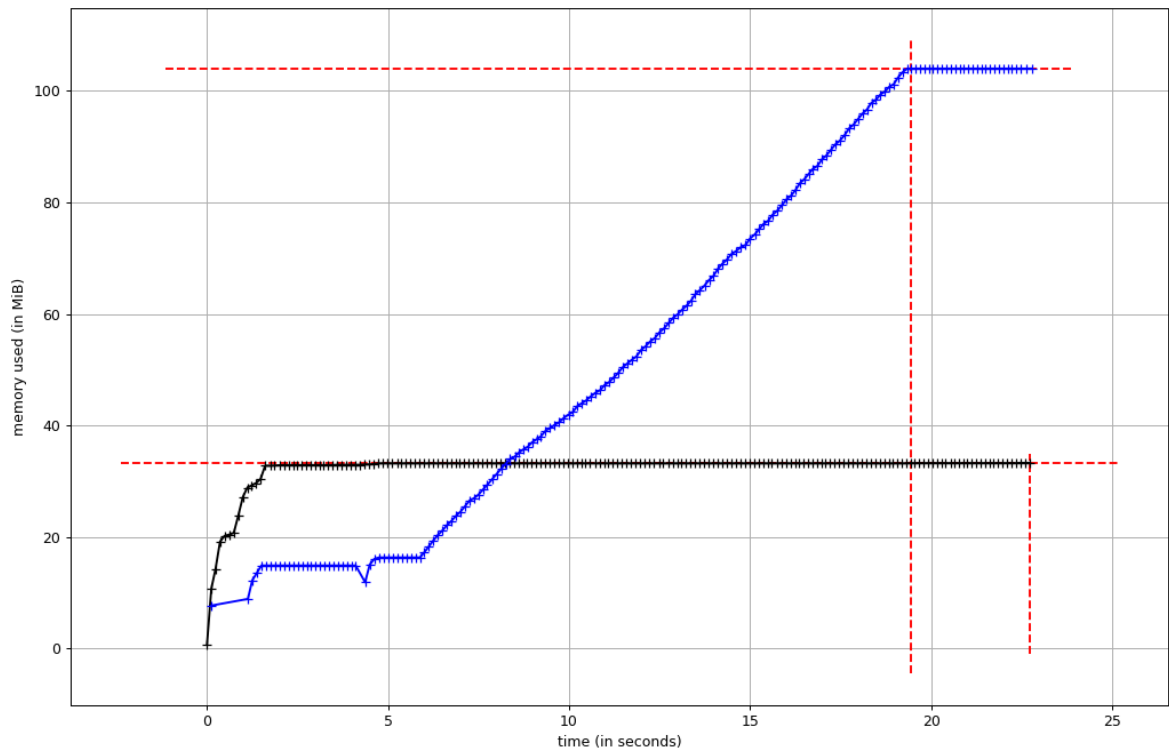


Figure 18. Memory usage by omniscient debugging Thonny – Sudoku.

5. Efficient Fibonacci

The developer is to write a more efficient Fibonacci number calculator.

Exercise no 7 at http://progeopik.cs.ut.ee/11_rekursioon.html#ulesanded

```
def fib(n, arvud={}):
    if n == 0:
        return 0
    elif n==1:
        return 1
    elif n in arvud:
        return arvud[n]
    else:
        arvud[n-1] = fib(n-1)
        return fib(n-1) + fib(n-2)

print(fib(10))
```

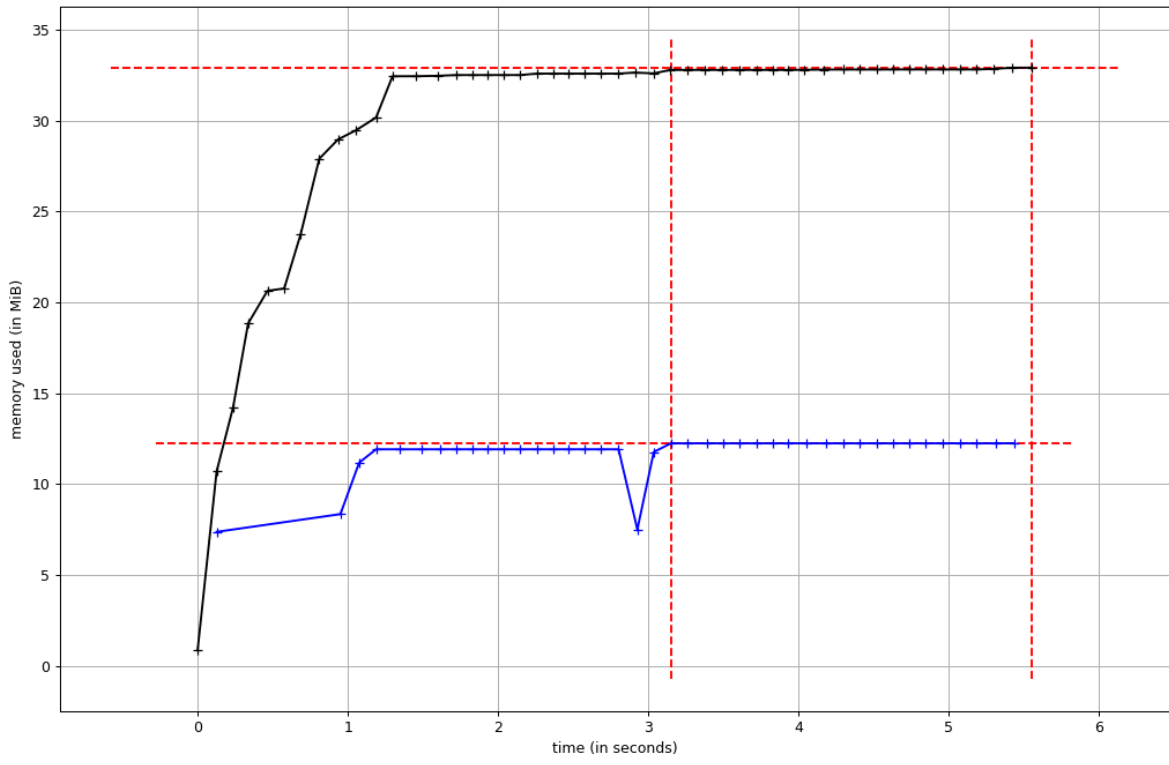


Figure 19. Memory usage by pre-omniscient debugging Thonny – Efficient Fibonacci.

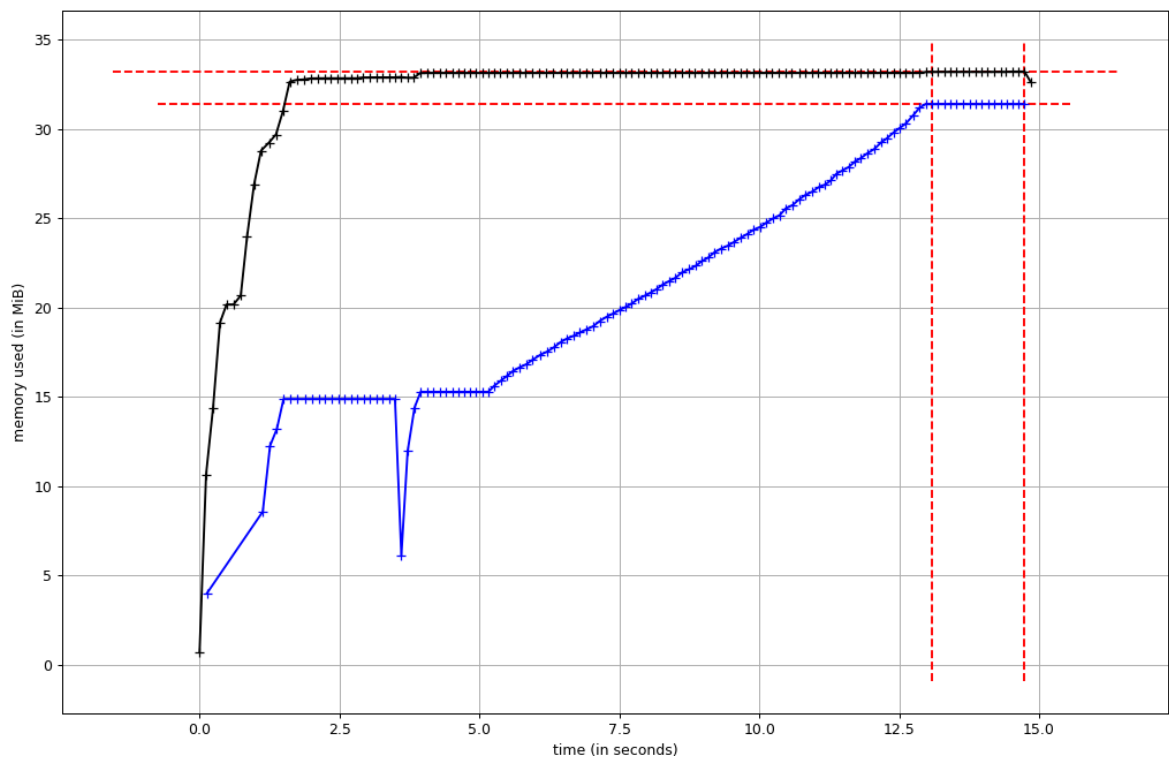


Figure 20. Memory usage by omniscient debugging Thonny – Efficient Fibonacci.

6. Ancestry Check

The user must write a recursive function checking whether the person in the functions first argument is an ancestor of the second.

Exercise no 9 at http://progeopik.cs.ut.ee/11_rekursioon.html#ulesanded.

```
def on_eellane(esiisa, inimene, sõnastik):
    if inimene in sõnastik:
        if esiisa in sõnastik[inimene]:
            return True
        else:
            return on_eellane(esiisa, sõnastik[inimene][0], sõnastik) or on_eellane(esiisa, sõnastik[inimene][1], sõnastik)
    return False
f = open("sugupuu.txt")
sugupuu = {}
for rida in f:
    rida = rida.strip()
    rida = rida.split()
    rida[1] = rida[1].strip(",")
    sugupuu[rida[0].strip(":")] = rida[1:]
f.close()
print(on_eellane("Pjotr", "Siim", sugupuu))
```

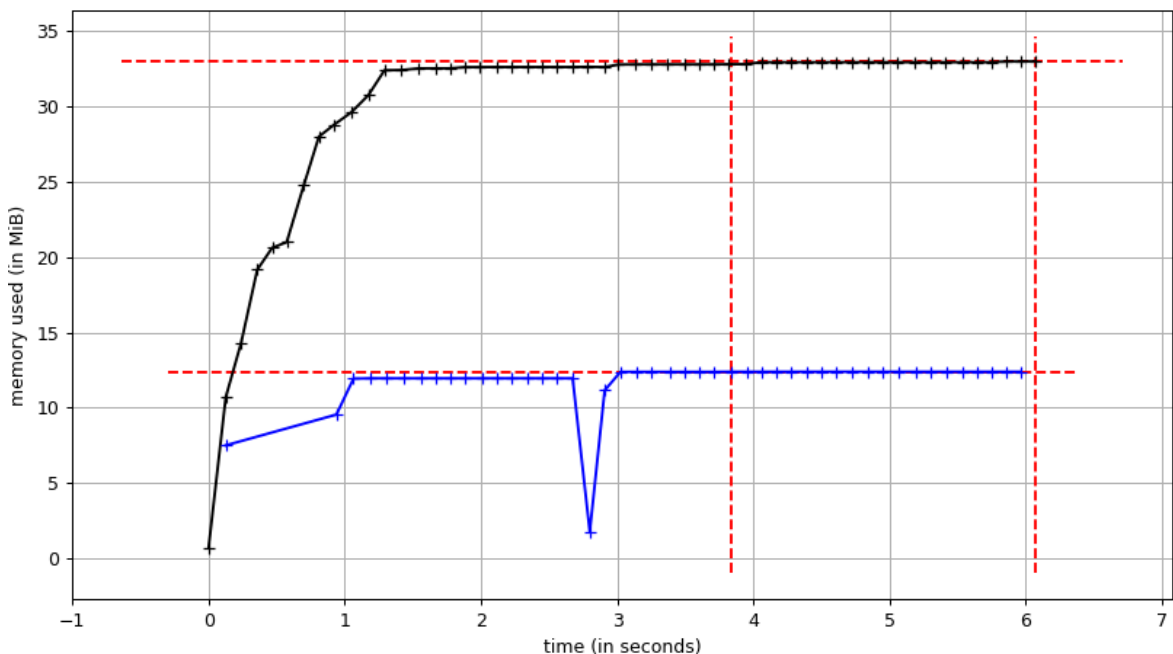


Figure 21. Memory usage by pre-omniscient debugging Thonny – Ancestry Check.

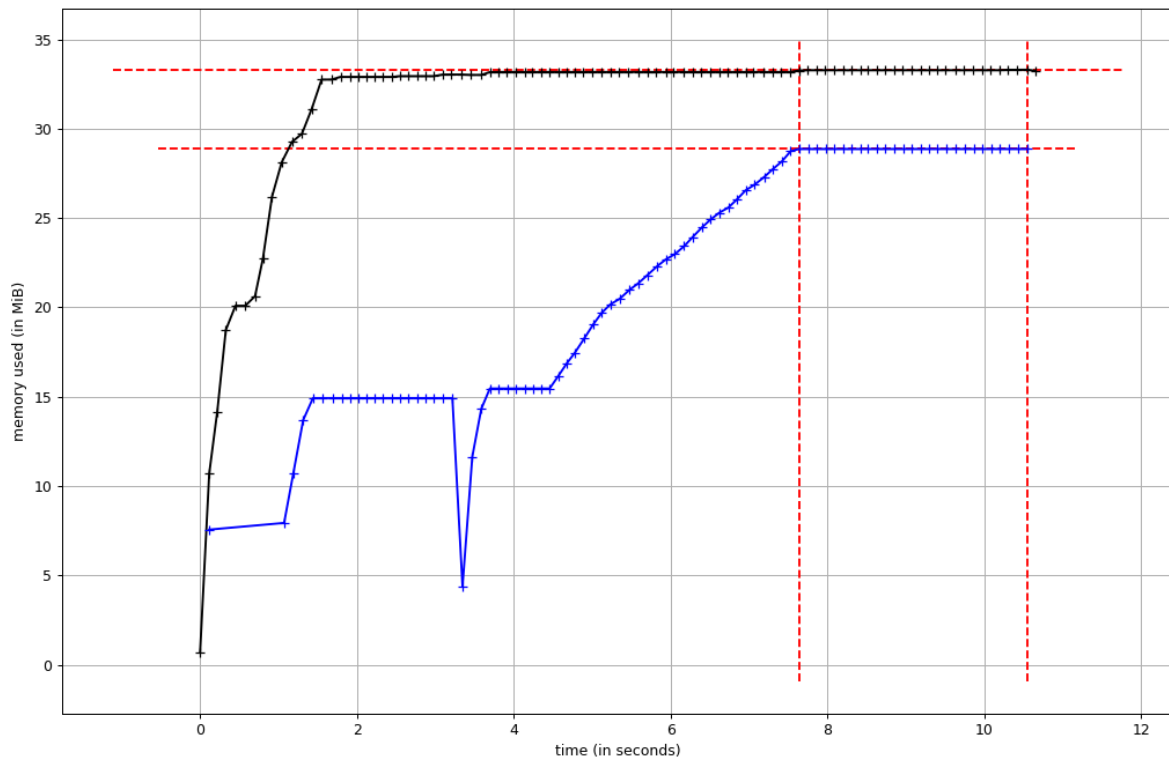


Figure 22. Memory usage by omniscient debugging Thonny – Ancestry Check.

7. Nearest points (bonus task)

An application is written that determines the 2 closest clicks made in the window. The calculation of the 2 nearest points was tested.

Exercise text:

“Kirjuta graafiline Pythoni programm, mis palub kasutajal teha programmi aknas 3 hiireklõpsu. Iga klõpsu peale ilmub hiirekursori kohale täpik, rist vms. Kui kasutaja on teinud 3 klõpsu, peab programm ütlema, millised kaks punkti kolmest on omavahel lähimad.

Kasutajaliidese tegemiseks soovitame kasutada Tkinteri raamistikku. Raamistiku kasutamiseks leiab esmased juhendid ja näited siit: <https://programmeerimine.cs.ut.ee/tkinter.html>. *NB! Kui oled seda lehte juba külastanud, siis tee lehele uuesti laadimine, sest 22. oktoobril lisati sinna üks uus näide, mis aitab antud tärnülesandega kiiremini pihta hakata.*“

```

from math import sqrt
def arvuta(punktid):
    kokku = len(punktid)
    kaugused_ja_vastavad_punktid = {}
    m = 0
    for punkt in punktid:
        for i in range(len(punktid)):

```

```

    esim_punkti_nr = str(punktid.index(punkt)+1)
    teise_punkti_nr = str(i+1)
    if punktid.index(punkt) == i:
        continue
    firstx = punkt[0]
    firsty = punkt[1]
    secondx = punktid[i][0]
    secondy = punktid[i][1]
    kaugus = sqrt((secondx - firstx)**2 + (secondy - firsty)**2)
    kaugused_ja_vastavad_punktid[kaugus] = esim_punkti_nr, "ja",
teise_punkti_nr
    vähim = min(kaugused_ja_vastavad_punktid)
    vähima_paari_nimi = kaugused_ja_vastavad_punktid[vähim]
    return vähima_paari_nimi

points= [[55, 42],
[538, 68],
[99, 238],
[549, 276],
[44, 381],
[554, 393],
[17, 517],
[481, 553],
[213, 565],
[223, 420],
[251, 211],
[255, 55],
[469, 163],
[456, 174]]
print(arvuta(points))

```

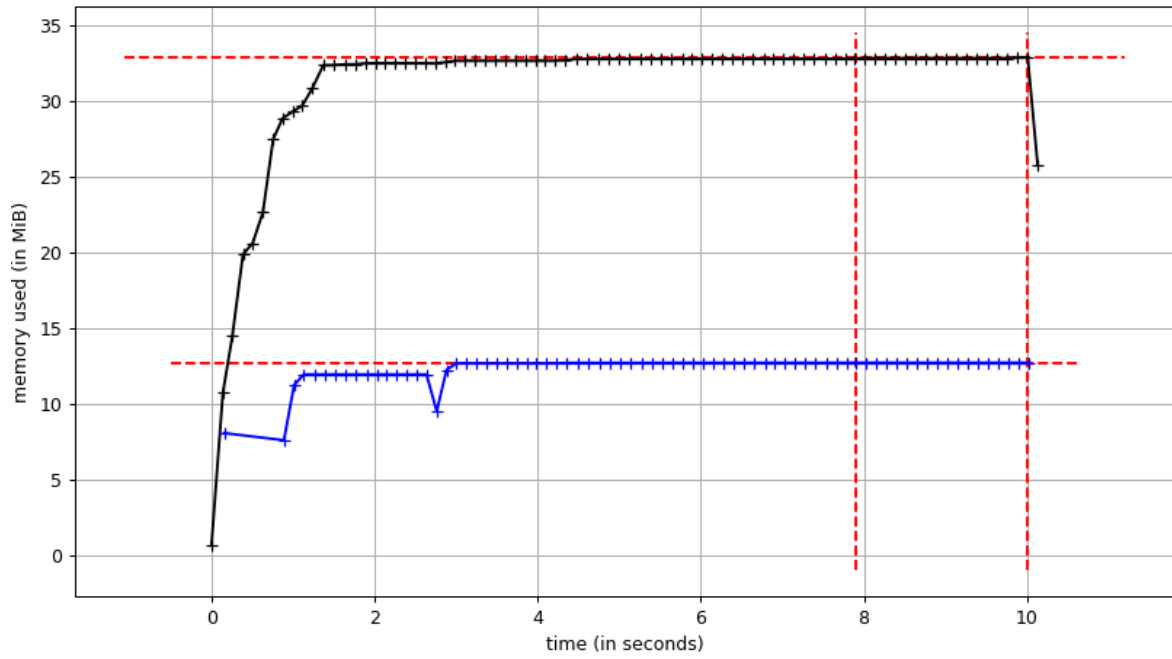



Figure 23. Memory usage by pre-omniscient debugging Thonny – Nearest points.

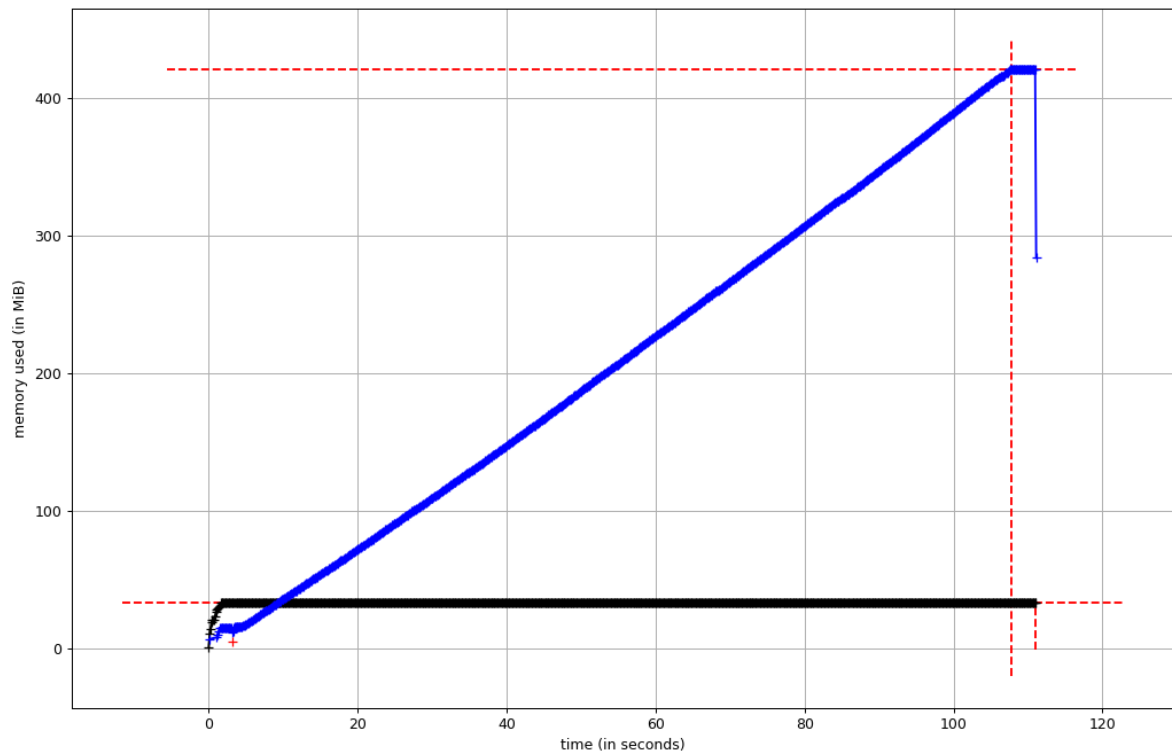


Figure 24. Memory usage by omniscient debugging Thonny – Nearest Points.

The analysis of the result is covered in section 5 of this thesis.

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Alar Leemet,

(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
 - 1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
 - 1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Omniscient Debugger for Thonny Integrated Development Environment,

(title of thesis)

supervised by Aivar Annamaa,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **14.05.2018**