RICE UNIVERSITY

A Browser-based Program Execution Visualizer for Learning Interactive Programming in Python

by

Lei Tang

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science

APPROVED, THESIS COMMITTEE:

Joe D. Warren, Chair
Professor of Computer Science

Scott Rixner
Professor of Computer Science and Electrical & Computer Engineering

Chris Jermaine
Associate Professor of Computer Science

Houston, Texas

April, 2015
A Browser-based Program Execution Visualizer for Learning Interactive Programming in Python

by

Lei Tang

Abstract

Good educational programming tools help students practice programming skills and build better understanding of basic concepts and logic. As Rice University started offering free Massive Open Online Courses (MOOC) on the internet, we developed a web-based programming environment to teach introductory programming course in Python. The course is now one of the top-rated MOOC courses, which is believed largely due to the successful web-based educational programming environment. Here we will introduce the thought processes behind the design and then focus on the key innovations incorporated in it. The main contribution of this thesis is an entirely browser-based Python program execution visualizer that graphically demonstrates the execution information to help students understand the dynamics of program execution. Especially, this tool can also be used to visualize and debug event-driven programs. The design details and unit test infrastructure for the program execution visualizer are both introduced in this thesis.
Acknowledgments

I would like to thank Professor Joe Warren, my advisor, not only for introducing the interesting problem to me but also for patiently providing careful guidance throughout this research. I also would like to thank Professor Scott Rixner for detailed technical advises, inspirations, and help on my work. I also thank Professor Chris Jermaine for serving as part of my committee and imparting helpful suggestions.
Contents

Abstract ........................................... ii
Acknowledgments .................................. iii
List of Illustrations ................................ vi

1 Introduction .................................. 1
  1.1 Massive Online Open Courses .............. 2
  1.2 Teaching Interactive Programming in Python .. 3
  1.3 Contributions ................................ 4

2 Web-based Programming Environment ........ 7
  2.1 Skulpt ......................................... 9
  2.2 CodeSkulptor ................................. 11
    2.2.1 CodeMirror ............................... 11
    2.2.2 CodeSkulptor Implementation .............. 13
  2.3 SimpleGUI: In-Browser Python GUI Library .. 15

3 Program Execution Visualizer .............. 19
  3.1 Online Python Tutor .......................... 21
  3.2 Viz Mode ..................................... 24
    3.2.1 Introduction to Viz Mode .................. 25
    3.2.2 Augmenting Skulpt ....................... 26
    3.2.3 Viz Mode Logger ........................... 31
  3.3 Viz Mode Application ......................... 32
    3.3.1 Visualizing Straight-line Program .......... 33
Illustrations

2.1 Skulpt Flowchart. Component A: Lexer breaks user code into tokens; Parser parses tokens to AST tree according to Python grammars; Semantics Analyzer analyzes AST. Special process Component B: Translate Python code to equivalent JavaScript code.

2.2 CodeMirror Example. Users load CodeMirror script and style sheet prior to use, then instantiate and attach an instance to a web component, a text area in this case.

2.3 CodeSkulptor Layout: A: Control button; B: CodeSkulptor editor; C: CodeSkulptor console; D: Demos and Documentations

2.4 Analog Clock Demo of SimpleGUI Library

2.5 Example SimpleGUI Frame

2.6 Example Games Using SimpleGUI Library. Users draw text, images, and geometric shapes on canvas; Control buttons are shown in BlackJack game (lower left). Timers may be used in games like Bunny’s Big Adventure (upper left) and Rice Racer (upper right). Key strokes or mouse clicks are used in all games.

3.1 Online Python Tutor Layout. A: In-browser program editor, click “Visualize Execution” to generate visualization panel; B: Program visualization panel, displaying user code(b1), program execution trace navigation widget(b2), cumulative output(b3), state diagram(b4); C: State diagram for visualizing object-oriented program
3.2 Online Python Tutor Program State Diagram (Left) and Execution Trace Entry Structure (Right) .............................................. 23
3.3 Viz Mode Layout. A: Viz mode control buttons; B: Viz mode console; C: Viz mode state diagram panel ........................................ 26
3.4 Viz Mode Control Model ................................................................. 30
3.5 Visualizing regular program using Viz mode ........................................ 33
3.6 Program execution utilizing break point feature ...................................... 34
3.7 Visualizing event-driven program using Viz Mode .................................... 36
3.8 Viz Mode Event Tags .................................................................... 36
3.9 Viz mode repaint canvas ................................................................ 37
3.10 Using Viz mode to visualize and debug Pong game program ............. 38

4.1 Comparing execution trace line number sequences based on string Edit Distance (Left) and Example Program (Right) .................. 43
4.2 Example Program (Left) and Execution Trace (Right) ......................... 44

5.1 Class Survey of CodeSkulptor’s Viz mode’s User Experience.
  Question 1: How often have you used CodeSkulptor’s Viz mode to help understand the behavior of your Python code during a typical week? Question 2: How helpful have you found Viz mode in learning the class material? ........................................ 48
Chapter 1

Introduction

*Sharp tools make good work.*

– Confucius (551-479 BC)

Teaching new programmers to program have been recognized as a pillar of introductory computer science education, and a great challenge at the same time. Compared to natural languages, programming languages are less human-understandable as they generate highly dynamic process from static text source code consists of branches, loops, recursions. To understand the underlying logic and mechanisms can be challenging to novice programmers. Hence, in the last decade, computer science educators have been passionately working on developing educational tools that help illustrate grammar and features of programming languages [1, 2, 3].

The conventional educational programming tools usually require users to install softwares with proper configurations and possibly more plugins or modules for different platforms. For example, students need to install IDLE (Integrated DeveLopment Environment) and Python libraries for learning Python, or to install different versions of JRE (Java Runtime Environment), Eclipse, necessary plugins, and proper configurations for learning Java etc. Moreover, since these configured softwares are running separately on users’ computers, any update to the operating system or the education tool itself would require every user to update their installation accordingly. These educational tools are widely used in traditional regular size classes where students
have access to the software and can seek help from instructors. However, when it comes to the large-scale online open course, the conventional programming tools are often too complicated and difficult to maintain considering large numbers of students with diverse backgrounds and in different geological locations.

In contrast, web applications that run in web browsers can easily support large-scale online education. The web application softwares are created in browser-supported programming languages. When users visit a relevant website, the web application user interface or client side code will be downloaded to the local machine automatically and runs in user’s browser. Besides static browser languages, using client side and server side scripting languages, databases, and cloud storages, people can create very powerful and interesting applications through web browsers. The developer can usually update the software on the backend without forcing users to make any change on their side. Therefore, web applications have shown its great potential in online education due to web browsers’ ubiquitous accessibility and cross-platform compatibility, such as online grading tool[4].

1.1 Massive Online Open Courses

Web applications start to appear in online education ever since the concept of Massive Online Open Course (MOOC) being proposed to take advantage of the modern internet. Since Massachusetts Institute of Technology (MIT) announced its OpenCourseWare Project in the year of 2001, a lot of attention from all over the world have been drawn to make education materials online and available to everyone [5, 6]. MOOC predecessors like Salman Khan and Apple Inc. started Khan Academy [7] and iTunes University [8] respectively by making tutoring videos and video lectures that are free to view and download. Those initiatives influenced most MOOCs offered
nowadays by many MOOC providers such as Udacity [9], Coursera [10], and edX [11].

A modern MOOC course has not only video lectures but also has homework, assignments, and quizzes. In order to better help students understand course contents, interact with instructors, or communicate with peers, the course designers usually need to provide students with different platforms. The platforms can be forums, blogs, social networks, and maybe specifically designed education tools such as a programming environment for a programming language course.

1.2 Teaching Interactive Programming in Python

Since 2012, Rice University Professors Joe Warren, Scott Rixner, and John Greiner start to offer free online course named “An Introduction to Interactive Programming in Python (IIPP)” on a popular MOOC website www.coursera.org. This course is now one of the most welcomed and top-ranked courses rated by both students and major MOOC course sites. Tens of thousands of students enrolled in each class session. As we look back, the success of this MOOC course is due to not only excellent course materials, but also the online programming environment created for students to use.

Prior to IIPP course, an internal version of the course at Rice, named “COMP 160: Introduction to Computer Game Creation”, was proved to be successful. In COMP160, the professors teach students to program in Python by creating simple computer games such as stopwatch, memory, blackjack, and Asteroids. We choose Python as the first programming language for novice programmers due to its ease of use. Python is much easier to learn compared to other programming languages. Its learning curve is gradual. It is an interpreted high-level language, open-sourced, and requires little effort to install, which is also object-oriented, extendable, portable, and
come with abundant libraries. Students at Rice learned to program in python, build exciting games, and enjoyed the course a lot.

How do we continue the success and teach large numbers of students efficiently in an MOOC course? The traditional in-person help from instructors won’t work in this case. Students may encounter problems from programming environment installation, configuration, debugging and so on. Therefore, what we really need is a ubiquitous programming environment. Besides the essential features the programming environment also needs to be easy to access and scalable, we also want it have more advanced functions, for example, to behave like a programming assistant that helps students to better understand programs as well as point to the possible problems in their own work. Also, we want it to be able to cloud save user’s code for review, modification, and sharing. Since we teach students to build Python games, the programming tools have to run in the browser to enable students to interact with the programs. Therefore, the programming environment has to be an internet-based web application that everything should happen in the browser including Python itself, Python libraries, and other functionalities. For this reason, we developed an integrated browser-based Python programming environment for learning interactive programming in IIPP, which will be elaborated in this thesis.

1.3 Contributions

In this thesis, we introduce an in-browser Python program execution visualization tool named “Viz Mode”. While the design of Viz Mode leverages many existing tools and ideas, such as it was built based upon CodeSkulptor and utilizes Online Python Tutor[12]’s program execution trace visualization code, Viz Mode also incorporates several novel features:
• Viz Mode visualizes interactive program execution in the browser. To support building Python games, we designed our online programming environment to enable students to write Python games in browsers. Viz Mode can also visualize the program interactions and help students better understand how event-driven programs work. To the best of our knowledge, this is the first Python event-driven program execution visualization tool.

• Viz Mode increases the limit of program length for execution visualization. Choosing not to use backend server, Viz Mode processes Python code and generate visualization data in user’s web browser so as to gain more computation power and avoid possible congestion on the server side. Therefore, compared to other server-based visualization tools, Viz Mode largely increases the limit of program length typically from several hundreds of lines to thousands of lines.

• We create breakpoint features for viz Mode to add more power to stepping manipulation. Besides traditional statement level navigation, users can step at event level for visualizing event-driven program in Viz Mode. Moreover, break points enable users to manipulate program execution visualization even more precisely, such as visualizing recursion, looping.

• Viz Model not only rolls back program run-time information, but can also roll back canvas content and incrementally repaint canvas components thus providing vivid explanations for stepping through event-driven programs.

The rest of the thesis will be organized as follows. Chapter Two introduces the web-based interactive programming environment, named Codeskulptor, and its application in our MOOC course. Chapter Three discusses the key innovation in this thesis – a program execution visualization tool, named Viz mode. We integrated this
tool into CodeSkulptor. In Chapter Four, we will introduce the unit testing method for Viz mode. Chapter Five then finalizes the thesis including future works.
Chapter 2

Web-based Programming Environment

A web-based programming environment is a web application for executing programs on a web server or user’s local machine. The same environment is accessible via a browser anywhere and anytime. To use a web-based programming environment is much more convenient than the typical process of downloading and installing a huge IDE such as Eclipse or Visual Studio, and then configuring the environment on your machine. Therefore, it is not surprising that recent years have witnessed a great number of web-based programming environments for various languages that allow users to develop, run, and even instantly share their codes inside the web browser.

Since they are convenient and simple-to-use, the web-based programming websites have become particularly popular in self-study and distance education [13]. Programming learners and students at school are typically not familiar with the advanced features of a programming language or its sophisticated IDE. Instead, web-based programming websites provide them an alternative way to fiddle their ideas. Up to now, a number of web-based environment for programming already exist such as Skulpt [14], jsdo.it [15], jsFiddle.net [16], repl.it [17], compileonline.com [18], Cloud9 [19] and so on. These web-based programming environments differ in the programming languages they support, the way they execute user code and other additional features such as saving and sharing program, online collaboration, open course or commercial. For example, Skulpt is designed specifically for running Python entirely in the browser. Some other websites focus on particular areas such as jsFiddle.net[16], a
javascript/HTML/CSS online editor. jsFiddle.net is a very popular tool used by many website developers to test quickly and share code snippets from inside the browser. There are also web-based programming environments supporting multiple programming languages, for example, repl.it [17] and compileonline.com [18]. Different from compileonline.com, repl.it runs user code in the browser and also support cloud saving and sharing coding session. Beside the functionalities mentioned above, there are also websites supporting advanced features including editing the same code at the same time and real-time chatting during coding session. For example, Cloud9 [19], it is a cloud-based commercial web-based programming IDE that provides the feature.

In the search for a good web-based programming environment for our MOOC course, we realize these websites share several common drawbacks when it comes to application in our course. Despite the fact that some of them does not support Python at all like jsFiddle.net, one of these drawbacks is that many websites execute user code on backend server. Therefore, it has relatively poor scalability and may be dysfunctional when running offline, like compileonline.com. We are looking for a completely browser-based python programming environment to teach large numbers of students. There are websites like repl.it satisfies this requirement yet lack ability to handle moderate-sized interactive python programs in our preliminary tests. Moreover, we are also expecting an environment with more functionalities to enable students to build interesting interactive python games. In the meanwhile, the cloud saving and sharing function is a necessary feature in our MOOC course. Unfortunately, none of the existing environment provides all features we need. We, therefore, decided to build our browser-based python programming environment based on open source tools like Skulpt.
2.1 Skulpt

Skulpt [14] is an entirely in-browser implementation of Python, where no server-side support is needed for executing Python program. The main idea of Skulpt is to translate user’s Python code into equivalent efficient JavaScript code that can be executed in the browser and generate the correct output or behavior as original Python code should do [20]. It has been heavily used in different places from programming education website such as How to Think like a Computer Scientist [21] and Problem Solving with Algorithm and Data Structure using Python [22] to MOOC course on Coursera named “Programming for Everybody” [23].

The core of Skulpt consists of a lexer, parser, and compiler passes in plain JavaScript that are partially ported from CPython [24]. Its workflow can be described as the following steps:

- Lexer and Parser break Python code into Python language tokens and build Abstract Syntax Tree for input program respectively;
- Semantic analyzer checks semantic errors to produce an intermediate representation of the program according to Python grammar;
- Code generator turns the intermediate representation into runnable JavaScript code, where imported modules and other built-in methods have to be first compiled to JavaScript in advance;

This approach was proved to have good scalability according to our own experiences in MOOC course as well as other user’s feedback and comments. The main steps are illustrated in the following flowchart:
Since Skulpt does not need backend server support and runs user code entirely in their browser, it avoids server side congestion so that make it more reliable and sustainable. It has relatively simple, and little-customized GUI application based on CodeMirror [25], which is an in-browser text editor. By using Skulpt, Skulpt JavaScript code and GUI application will be loaded and automatically launched in the client browser. It is possible to use such an application even if the network is offline, again since Skulpt works perfectly fine without server support.
Another merit of Skulpt is that it is an open source project, which means we have full control of both its functionalities and GUI application. Skulpt brings us hope to be able to customize our own in-browser Python programming environment with other functionalities based on Skulpt’s basic features.

2.2 CodeSkulptor

We believe that a good design of course and programming environment are the two main factors that influenced the success of an introductory programming course. With that thought in mind, we explored many existing web-based programming environments. Professor Scott Rixner then decided to build a new programming environment tool name http://www.codeskulptor.org and ensure it is fully functioned yet simple enough to use. This idea was greatly encouraged by Skulpt and has achieved great success in our several sessions of MOOC course.

CodeSkulptor improves Skulpt in several aspects including a more friendly user interface with intuitive buttons and commands, better layout, complete documentation for both CodeSkulptor itself and Python programming language, cloud saving and sharing feature, and other extensions which we will introduce in this session.

2.2.1 CodeMirror

An in-browser text editor is an essential part of web-based programming environment that enables users to develop their code as well as interact with backend server or local scripts by submitting the text in the editor area. Among many of those editors, CodeMirror is a widely used versatile in-browser text editor, which is implemented in JavaScript. The original developer of CodeMirror created an elegant core library that provides only the editor component and makes it fairly easy to use and reliable.
Normally, users only need to load its script and style sheet simply and create a CodeMirror instance into their web pages. For example, the following code embeds a CodeMirror instance into TextArea named “texteditor” in a web page:

```html
<html>
<head>
  ...
  <!--load codemirror script and stylesheet-->
  <link rel="stylesheet" href="codemirror.css">
  <script src="codemirror.js"></script>
  ...
</head>
<body>
  <form>
    <textarea id="text_editor">...</textarea>
  </form>
</body>
<script>
  var editor = CodeMirror.fromTextArea(document.getElementById("texteditor"), {
    lineNumbers: true,
    mode: "text/html",
    matchBrackets: true
  });
</script>
</html>
```

Figure 2.2: CodeMirror Example. Users load CodeMirror script and style sheet prior to use, then instantiate and attach an instance to a web component, a text area in this case.

As an excellent open source project, CodeMirror provides user rich programming API and css theming system so as to enable users to create their add-ons or customize their program editors. It currently supports over 60 programming languages and has many features and add-ons including various themes, code folding, configurable keybindings, programmable gutter, etc. This feature make CodeMirror widely used by SQLFiddle [26], pythontutor.com [12], jsfiddle.com [16], Bitbucket [27], Firefox and Chrome developer tools and so on. Therefore, CodeMirror was selected as program editor in CodeSkulptor.
2.2.2 CodeSkulptor Implementation

The motivation of building CodeSkulptor is to provide students with a reliable, easy-to-access online programming environment. It was named after its ancestors CodeMirror and Skulpt—a sophisticated in-browser program editor and a powerful/scalable in-browser Python engine.

![CodeSkulptor Layout: A: Control button; B: CodeSkulptor editor; C: CodeSkulptor console; D: Demos and Documentations](image)

Program editor is an important component of a programming environment that people will mainly use. CodeSkulptor implements a customized CodeMirror to be its in-browser Python program editor as shown in Figure 2.3 Component B. CodeSkulptor editor has many handy features including Python grammar highlighting, code folding, and many useful key bindings (such as comment/uncomment, indent, search). CodeSkulptor editor is maximized in the browser compared to other websites since it is a programming-oriented tool.

CodeSkulptor utilizes Skulpt as its core Python computational engine in the
browsers. The Skulpt engine parses user code instance and translate it into equivalent JavaScript code that runs in an encapsulated virtual machine in the browser. In addition, CodeSkulptor implements a subset of standard Python modules such as math, random, regular expression, time, urllib2, and collections, etc. In order to create interactive programs in our course, CodeSkulptor also provides students with three custom modules for graphics in the browser including simplegui module for building interactive programs and drawing, simplemap for drawing on the map, and simpleplot for plotting numeric data. User needs to import these modules prior to their code if needed. Skulpt engine will first build equivalent JavaScript modules for user imported Python modules before translating user code.

There are several user control buttons in Component A control area in Figure 2.3. Clicking the “Run” button will make CodeSkulptor first read user program in the editor, then input the program into Skulpt engine for processing. Any output that may happen will be shown in CodeSkulptor console, as shown in Figure 2.3 Component C. “Save” button saves the content of CodeSkulptor editor to google cloud-storage and shown associated URL in address bar pointing to the file on cloud-storage. When changes are made, user may click the “Refresh URL” button to save the new version and generate a new URL for updated version. “Download” buttons enable users to download their remotely stored code to the local machine. Conversely, local programs can be loaded via “Open Local” button. Intuitively, “Reset” button resets everything except user code in the editor, including console output, any pop-up windows, and any other internal variables and settings to default values.

In order to maintain good reliability and extensibility, CodeSkulptor was implemented following Model–View–Controller (MVC) software design pattern. It consists of three interconnected parts: a model for representing and processing internal infor-
mation including the callback functions of buttons and Skulpt; a view for presenting information or result to users such as to update status of GUI elements, display result etc. The central component controller coordinates model and view. By this means, view and model are very well decoupled. Therefore, updates happened in one side could be self-contained and will not affect the other side.

Since CodeSkulptor is built based on Skulpt, which is an entirely in-browser Python engine that translates and executes Python code in the browser, CodeSkulptor does not rely on backend server either. When the necessary scripts and style sheets being loaded in user’s browser, it can runs by itself without any help for outside except for cloud saving and sharing. This is very helpful for students with low internet connection rate or high internet service rate. Dr. John Greiner at Rice University Computer Science Department and his colleagues have built a very nice documentation link for both Python language and CodeSkulptor itself including its modules. Demos and Tips are also shown in Component D in Figure 2.3. All These together make CodeSkulptor a very popular tool that plays a vital role in our MOOC course.

2.3 SimpleGUI: In-Browser Python GUI Library

Last section introduced the way CodeSkulptor executes straight line program by reading user code from editor, processing and executing in Skulpt engine, and then displaying the result in the console. This design works well for straight line programs yet not for interactive programs with graphics and manipulations as described in our course design. Hence, a good Python in-browser GUI library that can be used in CodeSkulptor drawing graphics and enables users to interact with the program is a necessity according to our course design.
We explored a number of full-featured existing Python GUI library including TKinter [28], wxPython [29], and game-focused Pygame [30], etc. to see if we can port this library to browser. Those are all very popular GUI libraries that has been widely used. However, none of these libraries are design to support browser-based Python not to mention application in CodeSkulptor. Also, these libraries are production level libraries of moderately complex that might be overwhelming to novice programmers in our course. These facts lead us to our ultimate decision to build our in-browser Python GUI library named “SimpleGUI” indicating its function and easy to use.

SimpleGUI library provides user with a set of simple drawing commands ranging from creating canvas, drawing element on canvas to importing sounds and images, creating timers and so on. Similar to regular Python library, simpleGUI library can be imported at the begin of user code. User then will be able to create a canvas, buttons, input areas on a pop-up windows that enable users to interact with the program itself. Here is one analog clock example demonstrating the way we may use SimpleGUI library (Figure 2.5).

In this program, we first import SimpleGUI module and other necessary modules. We then use simplegui.create_frame to create a new frame with canvas and control areas on it (Line 32). User can load images such as clock background and hands using simplegui.load command to port images from their cloud storage URLs (Line 40-43). In order to draw elements on canvas, a draw handler is required and will be periodically called to refresh the canvas content (Line 21). In the above draw handler, it draws city name text and clock background images and hands with precise location, size, and angle, etc. In order to add more interaction, user can also create buttons and input area into SimpleGUI frame control area. The handlers of these buttons and input areas are defined prior to use. In this example, the “Houston” button callback
Figure 2.4 : Analog Clock Demo of SimpleGUI Library

function sets the time zone to be Central Time so as to adjust GMT time by -5 hours and assign CITY_NAME to be “Houston” that will be painted on canvas when canvas being refreshed (Left, Figure 2.5). Also, user may input another time zone in the text area to adjust the clock to show the current time in that area and show time zone text on canvas (Right, Figure 2.5).

This Python GUI library has been proved to be very popular in our course due to its simplicity and easy-to-use feature. Students have creates lots of interesting
games based on SimpleGUI library. Here are some examples in Figure 2.6.

Figure 2.6: Example Games Using SimpleGUI Library. Users draw text, images, and geometric shapes on canvas; Control buttons are shown in BlackJack game (lower left). Timers may be used in games like Bunny’s Big Adventure (upper left) and Rice Racer (upper right). Key strokes or mouse clicks are used in all games.
Chapter 3

Program Execution Visualizer

A good programmer needs to have good understanding of fundamental knowledge and a lot of coding experiences. Even if you understand the language and fundamental concepts of programming correctly, it still takes a long process before mastering programming. However, it is not always the case. Many novice programmers suffer the painful process of having continuously to correct their misconceptions they’ve built in their early programming age [31].

Students’ misconceptions of programming and related topics have been studied for some time. For example, Mayers [32], Bonar and Soloway [33] try to understand the source of misconceptions based on mental models. Spohrer [34] and Pea[35] etc. studied on semantic and natural language factors. This research result shows students’ various misconceptions falls into the falling categories:

- Memory usage, references, and pointers;
- Program language syntax and grammar;
- Control flow such as if-statement, switch, loops, conditions, continuations, exceptions;
- Function call and recursion;
- Object-oriented programming concepts and usage;
- Other miscellaneous issues.

Since program execution is highly dynamic and hard to inspect, it buries many misconceptions to novice programmers, which hinder them from making progress to-
wards further study and advanced topics. Therefore, many computer science educators programming visualization tool tries to help in understanding the basic concepts and different phases during execution [1, 3]. The basic idea of the program visualization tool is to make abstract thing visible such as variable values, heap, references diagram, call stacks, and maps static textual representation to highly dynamic process [36]. Sorva comprehensively discussed over 40 such visualization tools in his Ph.D. dissertation [37], which all consist of several major components including one GUI pane shows the source code with the the line highlighted which is about to be executed; another pane shows a visual representation of the run-time state (e.g., stack frame contents, heap objects), and control widgets allow the user to step forward and backward through execution points. Teemu and Sorva then implemented their program execution visualization tool called “UUhistle” [38] as a Java web-start application or applet requiring pre-installed Java 6 and above. “UUhistle” visualizes program run-time states and in the meanwhile, allows students to play the roles as a computer to execute the program step-by-step by themselves and compare the process with computer generated intermediate results. “UUhistle” has been successfully used at Aalto University in Finland and open to public now like some other tools. For example, Philip Guo [39] created an open-source web-based tool “Online Python Tutor” to help students understand the process of program execution, which turns out to be a very popular program execution visualization tool used by many educators from universities and MOOC course instructors.

In order to help students better understand their code and correct possible misconceptions, we would like to incorporate program execution visualization tool into our course programming environment. Therefore, students may turn to visualization tool for help most of the time. This self-learning experience can be views as a good
substitution of instructor’s over-the-shoulder help, particularly in distance education. Due to various limitations and drawbacks of existing programming visualization tools, after we explored many tools and particularly Online Python Tutor, we eventually made a decision to build our programming visualization tool embedded in CodeSkulptor. In the following sections, we will mainly discuss the motivation, implementation, features, and application of our new visualization tool named “Viz Mode”[40].

3.1 Online Python Tutor

The “Online Python Tutor (OPT)”[12] is a Python program execution visualization tool that runs within web browsers without the need to install any extra software or plugin. OPT hosts its frontend OPT website written in HTML, CSS, plain JavaScript and libraries on Google App Engine platform (Python 2.7) and uses backend server running Python to generate an execution trace for “Dynamic Visualization” [41]. The ease of access makes OPT a very popular program execution visualization tool in CS education and self-learning. Using OPT is as simple and easy as visiting a regular website. Its GUI contains two main parts associated to two phases: Edit and Visualization phases showing in Figure 3.1.

Users usually type in their code in the in-browser program editor and then click the button “Visualize Execution” to submit user program to backend server for processing. OPT backend takes user programs as input and produces an execution trace as output. Backend server executes user programs using standard Python debugger module (bdb), which stops execution after every executed line and records the program’s complete run-time state. The execution trace is an ordered list of entries of the run-time state that records up to the line of code is about to execute.
Figure 3.1: Online Python Tutor Layout. A: In-browser program editor, click “Visualize Execution” to generate visualization panel; B: Program visualization panel, displaying user code (b1), program execution trace navigation widget (b2), cumulative output (b3), state diagram (b4); C: State diagram for visualizing object-oriented program

The complete run-time program execution information were encoded into program execution trace in JSON format [42] and will then be returned from backend server to frontend. Figure 3.2 illustrates the structure of program execution trace entry that encoded the following information:

- The line number of code that is about to execute;
- Dictionary representation of variables and their current values;
- Call stack information: an ordered list of stack frames containing local variables
and their current values;
- The current heap elements or compound objects and their values;
- The cumulative output up to this execution point;
- Other information including current entry’s type, such as step line, return, exception, function call and so on.

This output execution trace will be parsed and ported into front-end visualization code made of two main JavaScript libraries: D3 [43] and jsPlumb [44], the former library is used to create corresponding HTML element for trace objects, jsPlumb is used to generate landmark points and arrows for visualization. The state diagram shows variables in frames stacked vertically where compound objects including class and object are shown in tables pointed by arrows (Figure 3.1). Current active frame/scope is highlighted in light blue as shown in Figure 3.2.

Figure 3.2: Online Python Tutor Program State Diagram (Left) and Execution Trace Entry Structure (Right)
OPT graphically illustrates dynamic program execution trace information that turns out to be very helpful and popular among novice programmers. Many universities such as UC Berkeley, MIT have used this program visualization tool in their CS1 courses to help students. In the meanwhile, hundreds of thousands of students on the internet have used this tool.

Unfortunately, we realize OPT has some drawbacks when it comes to application in our MOOC course. For example, it uses backend server to execute programs and generate an execution trace. Due to server’s limited bandwidth, memory, CPU capacity, it may turn out to be a bottleneck when there are many students trying to access to it at the same time especially on the due days of assignments. Secondly, it restricts the program to be shorter than 300 lines to guard against too much consumption on the server side. Thirdly, it requires internet connection all the time and might be slow and even dysfunctional when lost connection. Lastly but not least, it does not support interactive programming and GUI library like SimpleGUI. These drawbacks limits the use of OPT in our course and lead us to the final decision to create our program execution visualization tool, which is the key innovation in this thesis.

3.2 Viz Mode

In sight of these issues as well as the popularity of CodeSkulptor designed by Professor Scott Rixner, the author decided to integrate Python Tutor’s capabilities into CodeSkulptor. It has the same philosophy as OPT’s, which takes user program as input and generate execution trace for it, then generates state diagrams and renders them in the browser. Instead of using backend server, our approach was to add a new mode to CodeSkulptor. When the new mode is turned on, it will augment
original Skulpt code and instrument the compiled Javascript code. Executing this instrumented Javascript code would then not only evaluate the corresponding Python program, but also generate an execution trace similar to OPT’s execution trace for the program. We could then use Python Tutors frame/object diagram visualization capabilities (available in open source form as Javascript) to display a selected state in this trace. We call this tool “Viz Mode” (http://www.codeskulptor.org/viz/index.html).

3.2.1 Introduction to Viz Mode

To switch from regular mode to Viz mode, users may click the leftmost “Wrench” button in the circled area in Figure 3.3. This action will not only enable augmenting Skulpt but also change the UI. The code editor remained unchanged yet had more features, such as similar to OPT’s red arrow in Figure 3.1, this statement that is about to execute is highlighted in light blue.

The original console was splitted into two parts. Viz mode console (upper right) in B, is used to print the event names and output of print statements. The outputs prior to the statement to be executed is in black, and the other outputs afterward are shown faintly. After running each program, Viz adds two default events tags named “Begin” and “End” into Viz console. Similar Event tags such as button callback functions “houston” in Example 2.5 will be shown as “houston()” in Viz console. One event tag will be highlighted if currently highlighted statement is within that event function body. The four yellow framed buttons to the right of “Wrench” button were spawned to enable users to navigate through execution trace at both event level and statement level. Component C in Figure 3.3 is used to place the program state diagram.
In order to generate program execution trace, we first need to augment Skulpt to collect complete dynamic state information, and then encode the state information in Viz logger. Thirdly, when user navigate through program execution trace, Viz should parse execution trace and generate state diagram, and coordinate it with the highlight bar of use code, event tag in Viz console, and even canvas contents on the pop up frame for interactive programs. Similar to CodeSkulptor, Viz mode module is also designed following MVC pattern that consists of a VIZModel, a VIZView, and a mini controller VIZController.

3.2.2 Augmenting Skulpt

We augmented Skulpt at three levels: inserting regular recording lines that call Viz logger, creating new variables and fields used in generating trace and adding more
control mechanisms used in visualization. Since the actual changes are too many to be covered in this thesis, we only list some examples for each category to explain the basic idea:

1) Inserting Recording Lines

As introduced in Figure 2.1, Skulpt processes each line of code and recognize it as a proper type of statement. Then Skulpt internal compiler will compile each statement to generate corresponding JavaScript code. For general cases, we only need to insert trace recording lines right after each line of generated JavaScript code, such as `print` and `assignment` etc., to captures the complete run-time state information, such as:

```plaintext
skulptTrace = VIZ.VIZModel.GenSkulptTrace($gbl,lineno);)
vizTrace = VIZ.VIZModel.GenVizTrace($gbl,lineno, CS.view.outputnum, drawEventNo);
VIZ.VIZModel.CombineTrace(skulptTrace, vizTrace);
```

Where $gbl$ contains all the global variables, objects, and complete call stack information that can be used to generate stack frames. $lineno$ is current line number. CS and VIZ are instances of CodeSkulptor and Viz respectively. Different from OPT’s trace, the program execution trace combines the following two parts: Skulpt trace and Viz trace. The former trace is very similar to OPT’s trace called `skulptTrace` including python program run-time status while the latter one encoded more information used for Viz GUI elements like `outputnum` and `drawEventNo` used for highlighting Viz console output and repainting `simplegui` canvas components.

Usually we record runtime information after executing one simple statement, such as assignment statement. However, for some special statements, we might need to insert more lines into the parsed JavaScript code to record runtime information, for example:
a = [i for i in range(4)]

For this list comprehension example, Skulpt translates it to four steps in JavaScript. In order to watch the complete process of building list [0,1,2,3], we need to build one execution trace entry for each step. Therefore, more recording lines need to be added. Same thing for many other advanced cases such as nesting function calls, recursive calls, simple statement, yield, generators.

2) Creating new variables and fields

In order to provide detailed execution information, sometimes we may need to create new intermediate variables and new fields for objects used by Viz mode that capture dynamic state information. For example, in the list comprehension example above, to shown the list comprehension build process, besides adding more recording lines, we also created a new temporary variable named _tmp and added it into skulptTrace. This temporary value stores the temporary list during the build process and will be automatically removed from skulptTrace entry when the process finishes. Similarly, in order to show the return values of function calls, we also added a temporary variable named _return_ that stores the return value if any or None otherwise. Function definition line numbers are recorded in a new variable Sk.funcline, where Sk is the instance of Skulpt:

Sk.funcline[harshFuncName] = defLineNo;

In an interactive program, we may also want to watch canvas roll back along with statement level execution trace navigation. Therefore, we record all argument values while we call SimpleGUI library draw commands:

canvas.draw.circle(center_point, radius, line_width, line_color)

This command draws a circle on canvas, and its center_point, radius, line_width, and line_color are all recorded in a new field VIZ.VIZModel.drawOpArgs. These values will be used to repaint canvas so as to roll back canvas to specific point in program execution.
In other cases, we may need to add new fields to Skulpt objects, such as a statement instance, function object, etc. For example, in order to facilitate encoding function object and function name for current entry as shown in Figure 3.2, we added two new fields `funcname` and `funcvarlist` into function object, which contains function name string and argument list respectively. In the following example:

```python
def foo(a): a = a + 1; a -= 2; print a
```

The function body is defined as simple statement and need to be identified and treated specially since in Viz mode, we will need to break it into three simple statements instead of stepping over it as one single compound statement. Therefore, when doing semantics analysis, we added a new field `isSimple` in the function object to label it.

### 3) Adding control mechanisms

To better visualize the dynamic process of program execution in Viz mode, we need to coordinate every part during navigation. For example, we need to record current line number before stepping into a new scope so that can be popped out and resumed. Therefore, we create a `_numstack_` in `$gbl` to record these line numbers:

```java
$gbl._numstack_.push(Sk.currLineNo.toString());
```

Besides, we also need to control when and where to generate execution trace entries for loops and recursive function calls. As we introduced in Figure 3.3, we should highlight correct event tag in Viz console while the highlighted statement in Viz code editor is inside that event. In the same time, correct output text that printed out right before the highlighted statement should show in the Viz console. Moreover, the canvas content should also be updated according to the current position in the program execution. We implemented these using the method described in the following figure:

The figure above illustrates the following mechanisms: first, we record corresponding line number for each line of output in Viz mode, at certain point, we only show output text whose corresponding trace entry index is smaller than current trace entry index, meanwhile
faintly show other text after that point; Secondly, execution trace entries are grouped as a series of events including `BEGIN` for initialization and `END` pointing to the end of the entire trace. All the other trace entries of events fired after initialization will be appended to the end of the trace. For example, if current event is `SAj`, then event tag `EeventA` should be highlighted in Viz console. Similarly, canvas draw operations are grouped as draw events, if current highlighted statement is `SAi`, which is larger than `drawOp_m` yet less than `drawOp_{m+1}`, then we need to redo every draw operation in draw `Event1` before this current highlighted statement. However, for `SAj`, we need to redo the draw operations in draw `Event2` which are before entry No. `SAi`.

To make Viz mode more convenient to use, we also designed "Break Point" feature to enable users to stop wherever they want to inspect. This feature is quite handy especially for
visualizing or debugging loops/recursive function calls. The user can either add breakpoints by clicking the program editor gutter before execution or dynamically add more breakpoints during visualization. Since each program execution trace entry contains the program line number, we added a new field in VIZ.VIZModel called `breakPointLines` to keep track of all the line numbers user clicked to set a breakpoint. Based on the program line numbers, we can find out and order the execution trace entries (breakpoints). The user may click “run” or “Event Navigation” buttons to run a program in the event level back and forth till it hits the first breakpoint or entry, corresponding to a program line number in `breakPointLines`.

### 3.2.3 Viz Mode Logger

Similar to Online Python Tutor’s logger on the backend server, we also built a Viz model logger that runs in the browser and encodes program dynamic state information into execution trace. In order to include both python program run-time information and also Viz mode control/GUI information, the logger was built containing two main parts.

The first part takes program variables stored in `$gbl` introduced above as input. Global variables and objects in `$gbl` are encoded in the default frame on top of all other frames. Other call stack information stored in `$gbl.__stack__`, which is a stack of `$loc` that contains local variables and objects. Viz logger therefore creates call stack frames vertically based on `$gbl.__stack__`. Compound objects are encoded recursively that every compound object was assign an unique number used for generating variable references. This part of trace is also called “SkulptTrace” as it is mainly for storing program execution information. Similar to Figure 3.2, “SkulptTrace” encodes program execution run-time information stored in `$gbl` including global variables and objects and call stack information, such as `ordered_globals`, current `func_name` etc., where primitive types and objects are encoded differently, one as string and the other as heap object. The call stack information stored in `$gbl.__stack__` will be encoded as `stack_to_render` frames, which grows from top to button. All execution trace is built in JSON format [42].
In the meanwhile, the other part of Viz logger encodes other information such as output text line numbers, draw event number, and so on, named “VizTrace”. The information is also encoded in JSON format in “VizTrace”, which will then be used to control GUI and coordinate different components of Viz mode as introduce above.

To sum up, Viz mode is an OPT style program visualization tool embedded in CodeSkulptor. Unlike OPT, Viz does not support of backend server. Instead, it runs Python programs completely in the browser hence enables us to build program execution trace locally in user’s browser. This design brings us many advantages in practice for example: relatively good scalability compared to OPT. Skulpt is able to handle moderately-sized Python programs with thousands of lines compared to only 300 lines in OPT. Secondly, it generates execution trace and stored in users’ local machine which is faster than OPT since OPT need to send code to server, consumes large amount of memory on server and retrieve the data via internet. Therefore, this perfectly handles the server congestion problem automatically. Moreover, it can greatly reduce the time user needs to stay online, since it works perfectly whenever it is loaded in user’s browser and requires no more access to internet afterwards except for generating hash URL and sharing code online.

### 3.3 Viz Mode Application

Viz mode have been used to help MOOC course students to understand and debug straight-line programs and event-driven programs. The straight-line program does not have interactions with users, which will be compiled into equivalent JavaScript code and executed all at once. However, event-driven programs contain event callback functions that are compiled and will only be executed when users trigger its bonded event. The following two sections introduce the usage of Viz mode for visualizing the two types of programs.
3.3.1 Visualizing Straight-line Program

Straight-line programs are executed right after user click the “run” button. Viz then generates entire trace for this execution with a BEGIN and a END tag in the console attached to the beginning and end of the execution trace respectively as shown in Figure 3.5. We can use the event-level navigation (rightmost two yellow framed button) buttons to move execution trace entry cursor to the beginning of the entire execution trace. The tag of the current event is highlighted using light blue bar. Similarly, we also use light blue bar background to highlight the statement that is about to be executed. Then we can use statement level navigation buttons (the two yellow framed buttons to the left of event-level navigation buttons) to start walking through the execution.

![CodeSkulptor](image)

Figure 3.5 : Visualizing regular program using Viz mode

We walk sequentially through Line 1 to Line 22. Viz mode shows the program state diagram and variable references in the state diagram panel, in which we can inspect the
variable changes and call stack information. In Line 22, program entered scope of function bbb() which in turns called function aaa(). Viz built stack frames vertically from top to button and highlight the active scope frame in light blue color. In Figure 3.5, current highlighted statement is Line 14, where the trace entry cursor is located. Therefore in Viz console, we show the outputs of first two print statements in dark color since they were executed before the current trace entry. Particularly, the output of last print right before current trace entry is displayed in bold green font and underlined. In the meanwhile, the outputs of print statements after current trace entry are grayed out. This helps users clearly differentiate outputs before and after current position.

![Diagram of program execution utilizing break point feature](image)

Figure 3.6: Program execution utilizing break point feature

The program in Figure 3.6 recursively call `tupleSum` method to add up the first element in each tuple. User may be only interested in the values of `f` and `rest` in each recursive call. In this case, the user can click program editor gutter to set a breakpoint at Line 6 labeled as a red rectangle. When we click “run” button to execute the program, Viz will first run entire JavaScript version of the program and then search for correct position in the program execution trace to generate a visualization. If there is no breakpoint, Viz will
show the last entry. However, in this case, as the user has set a breakpoint at Line 6, Viz stops at the first entry with program line number 6 as it searches the program execution trace in order. After that, user may click event navigation buttons to navigate to previous or next occurrence of the breakpoint at the same program line.

### 3.3.2 Visualizing Event Driven Program

Visualizing Event-driven program is more complex compared to visualizing regular program due to the following more requirements: 1) Event-bonded callback functions will be executed only if users trigger the event, and Viz logger needs to log dynamically entire event execution trace, group the entries and append it to current program execution trace; 2) We shall coordinate statement and event level navigation; 3) Events including drawing statements on canvas require logging arguments of draw commands in order to enable canvas roll back during visualization. We revisit the analog clock example in Section 2.3 to demonstrate the way Viz mode visualize event-driven programs.

When we click the “run” button in Viz Mode, Viz immediately builds an initialization execution trace for this program with two event tags “Begin” and “End” in the console. Viz discovers simplegui canvas draw handler and simplegui timer in this program therefore it spawns two more buttons to the right of the two event-level navigation buttons, with a pencil and a clock icon on them respectively (Figure 3.7). Similarly, the same set of draw and timer buttons are created on the simplegui frame, located in the original mouse click and key strike echoing area. These two buttons enable users to manually fire canvas draw events and timer handlers, which are usually automatically fired in regular mode at high rates. Therefore, in Viz Mode, users have precise control over all events and can visualize the event-driven program in a fairly interactive manner. If there are more than one timers in the program, Viz builds drop down and pop up timer menus on the main frame and simplegui frame respectively which will automatically display when user hover their mouse on top of the timer button. Program timer `clock_timer` in this case is used to switch time
zone between Houston and New York every three seconds, and is shown as green “<Timer>” in state diagram since Line 52 started it. On the contrary, Viz uses red color to indicate the timer has been stopped.
After program initialization execution, we then fired a series of events as shown in Viz console Figure 3.8. We first show Houston time (Figure 3.9, Panel B) by clicking “Houston” and then “draw” button, which will attached two event tags. Similarly, we then show the time of New York (Panel C) and Time Zone 3 (Panel D). By clicking the last “draw” event tag, we jump to the beginning of the last draw event with an empty canvas as shown in Panel A. After executing Line 32, canvas turned to be Panel E, in which Viz first retrieve the logged CITY_NAME as an argument in simplegui.draw_text to redraw text on the canvas. Similarly, Viz mode redraw F through I after executing Line 33-36 respectively. This shows Viz can roll back canvas and repaint it along with current program execution trace.

Here is another interesting example. In one of our course mini-projects, we teach students to build their own versions of one popular arcade video game – Pong. Their program ought to allow users to move the two pedals to bounce the ball back and forth as shown in Figure 3.10. However, many students have trouble to write code that bounces the ball
correctly on the pedals and walls. Unfortunately, the original `simplegui` code refreshes and updates the ball position too fast therefore it is not possible for students to figure out what the problem is. However, using Viz mode, they are able to update ball position and in the meanwhile read the ball velocity step by step by clicking the “draw” button until it is about to hit the wall. This feature helps them check if the position and velocity being correctly updated by bouncing code.

From the examples introduced above, we see Viz mode provides a lot more control over event-driven programs thus improving users’ understanding and helping them debug their code.

Figure 3.10 : Using Viz mode to visualize and debug Pong game program
Chapter 4

Unit Testing Infrastructure for Viz Mode

Computer program bugs come with source code by nature, even in a small piece of code. One example is the famous binary search, according to Knuth, the first bug-free binary search program did not appear until 1962, 16 years later after it was published in 1946 [45]. As computer programs grow more complex, bugs become more and more frequent and difficult to discover and fix. Therefore, a variety of software testing methods have been developed to check the functionality, correctness, and locate bugs.

Among these testing methods, unit testing [46, 47] has become a widely used method. As a bottom-up testing approach, its goal is to test each independent unit of the program and verify the correctness of individual units. For example, by testing functionally isolated functions, classes, modules, unit testing can reduce the uncertainty of integration testing for big systems. This procedure includes first designing specific test cases for individual units including input and also standard outputs so that we can quickly find problems of independent units by comparing the outputs. These test cases accumulate through the development process to be a test suite. These tests can not only help identify bugs early in each isolated functions, but also be frequently used to verify the individual units via an automated process in the whole development cycle. If some unit tests fail after we made modifications, it clearly shows bugs may be introduced in this update. Therefore, unit testing can be used to verify the correctness of both basic individual units and integrated system.

Besides, the unit testing method also has other advantages in software development process. One such advantage is that it help us to write code that is easy to test. In order to
be able to support the particular type of testing for this very function, functions should be
developed small and focus on a particular operation instead of a lot of different processes at
the same time. This strategy helps reduce the chance of errors and improve code reusability
at the same time. Also, since developer can first design test cases in order to make sure the
basic units functions as expected, these test cases may be used as a blueprint of functions,
methods and can be used to verify if the desired function has been implemented or not.

Same as other testing methods, unit testing can not catch every error in the program
especially it tests only the correctness of basic independent units and not the integration
errors. Therefore, it is usually used along with other testing techniques. However, unit
testing is still a very useful tool for testing big system consisting of smaller components
such as Skulpt and our program execution visualization tool.

4.1 Unit Testing Skulpt

Designed as an in-browser Python program executor, Skulpt needs to support all language
features of Python. In order to make sure Skulpt implements these features both completely
and correctly, a testing program suite were built during the development period. According
to Figure 2.1, Skulpt executes input program in three main steps including parsing program,
translating Python program into equivalent optimized JavaScript code, and execution in the
browser. Therefore, we need to test all of the three steps thoroughly at both compile time
and run time in order to locate possible issues.

The Skulpt unit testing strategy follows the unit testing paradigm. To provide standard
outputs for unit testing, we first run the test programs in real Python. Using additional
scripts, real Python compiles each test program and dumps standard tokens, symbol table,
and abstract syntax tree into files with extension .expect stored on the local machine for
future comparison. Similarly, program outputs of print statements are also stored in a file
with extension .real. The next step after we generated standard outputs is to run Skulpt
itself and compare its intermediate and final results with standard ones. Hence, there are
five different types of comparison: \texttt{testTokens}, \texttt{testParse}, \texttt{testTrans}, \texttt{testSymtable},
and \texttt{testRun}.

However, Skulpt was written in JavaScript, which means the comparisons need to hap-
pen in JavaScript engine. Therefore, a virtual files system needs to be built for all test
programs and standout results with extension \texttt{.expect} and \texttt{.real} that they can be loaded
into JavaScript engine for automated comparison with Skulpt results.

This unit testing strategy was proved to be useful in practice. Especially more and more
test programs have been continuously added into test suite including programs designed for
testing language features and selected error-prone examples. These test programs can easily
trace issues during the development process and make sure the new changes do not break
the functionality of previously built units or discover the flaws of them for improvements.

4.2 Unit Testing for Viz Mode

In Section 3.2, we introduce the way we construct Viz Mode by augmenting Skulpt. We
set Skulpt flag \texttt{Sk.vizmode} to \texttt{true} to turn on Viz Mode, which will not only change the
way Skulpt compile user’s Python code, but also will generate very different JavaScript
code. Since the correctness of Viz mode execution should be prior to program visualization,
similar to Skulpt, we would like to build an unit testing strategy for Viz Mode as well.

As you may expect, unit testing Viz Mode is more complicated compared to testing
Skulpt alone. Since Skulpt is a pure program executor, that we only need to assure it
runs User’s program correctly. Besides the correctness of execution, Viz Mode should also
make sure program execution trace is generated correctly to reflect the order and dynamics
of program execution. Therefore on one hand, we turn on \texttt{Sk.vizmode} to unit testing
Viz mode by comparing its execution results with real Python’s standard results in both
compile and run time. On the other hand, we also need to compare program execution trace
with Online Python Tutor’s execution trace, since OPT’s execution trace is built based on Python debugger library (bdb) [48] which faithfully logs the sequence of execution in real Python.

In order to unit test Viz mode, we built Viz mode running in two modes: GUI mode and Non-GUI mode. Usually, Viz mode runs in GUI mode in the browser. Skulpt core and Viz mode module both need to interact with GUI elements during the calculation process, such as highlighting code lines sequentially, printing output to Viz console, generating program state diagram, and accepting user’s manipulations etc. However, this setting are not proper for automated testing upon a large number of test programs. Therefore, we built an alternative Non-GUI mode, which runs programs in V8 JavaScript Engine [49]. In Non-GUI mode, We created unit testing scripts to test the correctness of both programs results and execution trace in using command lines.

4.2.1 Execution Trace Correctness

Different from program results, program execution trace records dynamic information during program execution that are the most important data for program visualization. The trace needs to be tested thoroughly in terms of both trace entry line number sequence and also the trace entry content. The former part demonstrates the flow of program execution that may be very useful for understanding the order of execution, for example in recursions and yield. The trace entry content contains all information we are interested in, therefore, needs to be tested thoroughly. In order to unit testing Viz mode, we compare our program trace with standard trace taken from Online Python Tutor.

1) Checking trace entry line numbers

A program visualizer should first correctly demonstrate the execution flow of a program at a high level. Therefore, our first focus is to test the trace entry line numbers. As shown in Figure 3.2, in each of OPT’s trace entry, there is a field "line" indicating the line about to run. We then test each program in the test suite in both Viz mode and OPT,
generate an execution trace and extract the line numbers for each entry to create a sequence of line numbers. To capture the difference between Viz mode line number sequence and OPT sequence, we use a hash function to calculate a unique integer for each line of string containing the line number in the execution trace. Then, Edit Distance between the two transformed line number sequences will be calculated to show the dissimilarity between them. The Figure 4.1 demonstrates the basic idea. The comparison results show Viz mode trace missed one entry for Line 3, yet generated two extra entries on line 7 even if after the program execute `continue` statement. These results help us to locate and fix the problem.

2) Checking trace entry content

Program execution trace entry store program state information in JSON format, which will be used to generate state diagram and trace variable value changes. Therefore, we also compare Viz mode trace entry content with OPT’s trace by comparing each field.

Since entry content contains a lot more dynamic information, and Viz mode logs infor-

<table>
<thead>
<tr>
<th># Viz Mode</th>
<th># OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

![Figure 4.1](image_url)
mation in a similar yet not identical way compared to OPT, we cannot simply use String Edit Distance to calculate the difference between two trace entries. Instead, we compare trace entry as objects that contain different fields: as shown in Figure 3.2, some are simple fields, such as "func_name" and "line" etc. Therefore we can simply compare the strings for these simple fields. However, some other fields are complex, such as "stack_to_render" and "globals" etc. For "stack_to_render" we need to compare each frame in the stack. Moreover, for globals, we need to trace the compound data type down to primitive data by linking them together according to their "REF" number. For example:

```python
1 x = ("Rice","Univ")
2 y = {"1":"hello", "2":[3,x]}
3 z = [1,2,y]
4 print z
```

![Figure 4.2: Example Program (Left) and Execution Trace (Right)](image)

In this example, variable z can be expanded as the following form for comparison according to REF number:

```
["ordered_globals":[x,y,z],
 "stdout":",
 "func_name":",
 "stack_to_render":[],
 "globals":{"x":[REF,1], "y":[REF,2], "z":[REF,4],
 "heap":{"1":["TUPLE", "Rice","Univ"],
  "2":["DICT","1","hello"],
  "3":["LIST",3,[REF,1]],
  "4":["LIST",1,2,[REF,2]],
 "line":5,
 "event":"step_line"}]
```

Note here, the "REF" number might be different from the numbers in OPT due to the difference in two implementations. The unit testing for execution entry content is also made an automated process in the command line mode.

### 4.2.2 Viz Mode GUI Test

Although automated test for program results, the execution trace is fast and reliable, unfortunately, we can not use this method to test every component of Viz mode. Such as Skulpt
core, Viz module interaction with GUI elements, especially in interactive programs. Users will fire a sequence of events after the initial run of the program. The process is not easy to be tested automated. One reason is, GUI changes and manipulations are not easy to quantify and recorded for comparison. Moreover, different browsers and different versions may produce behaviors differently. The other reason is that, there are no standard results for unit testing the GUIs. As we’ve already known, OPT does not have SimpleGUI libraries we developed for building interactive programs, not to mention tracing the events based on it. Therefore, currently, the only way we used to test Viz mode GUIs is manually firing events and inspect the changes of both GUI elements including highlighting bars, buttons, and state diagrams. The test programs are collected through the development process, which are the error-prone programs we’ve ever seen to check specific functions of Viz mode, such as canvas roll back, background color changing. For each new test programs, we will manually build the standard checklist for unit testing them.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis describes a browser-based tool for visualizing Python program execution named “Viz Mode”. We use Viz Mode in Rice University’s introductory programming MOOC course and regular Rice courses. After running several sessions of these courses, we received many positive on-line reviews and feedbacks from our students indicating that our tool can improve the novice programmers’ understanding of the language features and fundamental programming concepts.

Viz Mode is an entirely browser-based program execution visualization tool. It works similarly to a web-based program execution visualizer called “Online Python Tutor” which uses back-end servers to process users’ Python programs. We integrated Viz Mode into CodeSkulptor – a web-based Python programming environment that provides easy access to users. CodeSkulptor runs user’s Python program by first translating it to equivalent JavaScript code and then executing the translated code in user’s browser to generate program results. As a special mode, when turned on, Viz Mode augments original CodeSkulptor translated JavaScript code to creates Python program execution trace. The trace consists of a sequence of entries that encode program run-time information after executing each statement. Then Viz Mode visualization code renders each entry in the trace to generate a sequence of program run-time state diagrams for users to navigate through. Since all computation happens in user’s browser, it nicely reduces both internet data transfer overhead and computation burden on the back-end server. Therefore, the up limit size of the
program that can be processed by Viz Mode is much larger than using a back-end server yet still depends on user machine’s computation power. Typically, Viz Mode can handle programs of several thousand of lines on modern computers.

Viz Mode provides various functions for visualizing program execution. In order to take advantage of OPT’s excellent visualization code to render state diagrams, we constructed Viz Mode program execution trace consisting of two parts: skulpt trace and Viz trace. Skulpt trace is designed to be identical to OPT’s trace for rendering, and Viz trace encodes other information Viz needs to provide more functions. Skulpt trace content will be rendered using OPT’s visualization code to provide a graphical view of program run-time information, such as references and call stack information. Each time user fires a new event while visualizing an event-driven program execution, Viz Mode generates a new sequence of trace entries and appends it to the end of current program execution trace. Therefore, in addition to statement level navigation, Viz Mode groups trace entries and enables users to navigate at the event level. Users can step in each event, watch status changes, and even repaint canvas contents. To the best of our knowledge, our tool is the first one that visualizes event-driven program execution. Viz mode also allows users to add breakpoints during visualization, which provides user precise control of where they want to stop and inspect.

Our in-class survey among over tens of thousands students shows that Viz Mode is very helpful to novice programmers. As shown in Figure 5.1, about 65% students use Viz mode every week. And 55% students in the entire class think Viz mode to be useful or very useful, which indicates 84.6% of all users who ever used Viz mode found it helpful to some extent. If we make a reasonable assumption that there exists strong correlation between people who use the tool often and those who think Viz mode to be very useful, we found over 93% students who often use the Viz mode tool (14% in Question 2 out of 15% in Question 1) like the tool and found it very useful in learning the class material.
Figure 5.1: Class Survey of CodeSkulptor’s Viz mode’s User Experience. Question 1: How often have you used CodeSkulptor’s Viz mode to help understand the behavior of your Python code during a typical week? Question 2: How helpful have you found Viz mode in learning the class material?

5.2 Future Work

For future work, we will keep improving the current programming environment to make it more stable, and particularly, make sure its browser-based Python interpreter is updated to match the latest version of real Python. We will also add more features to Viz Mode, such as to enable mouse hover function to inspect variable values; To make state diagram panel resizable so that user can hide it or expand it to view more details of state diagram. Since logging the execution process consumes memory, we will also focus on developing memory-efficient ways to build the execution trace. Moreover, we are planning to enable users to turn on/off the trace logger in Viz Mode dynamically. Therefore, if the users are
not interested in some events, they can turn off the tracer to avoid generating useless trace so as to save memory. We noticed the fact that this program execution visualization tool is now serving students not only from our MOOC courses, but also from all of the world. We will continue working on refining its GUI and adding advanced features to make it a popular tool.
Bibliography


