Bachelor

November 25, 2021

Interactive Command History Visualization for the REPL

Proof-of-concept implementation around the *Python* interactive mode

Nadine Muller

of Zürich, ZH, Switzerland (16-944-563)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor

Interactive Command History Visualization for the REPL

Proof-of-concept implementation around the *Python* interactive mode

Nadine Muller

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor

Author: Nadine Muller, nadine.muller@uzh.ch

Project period: 25.05.2021 - 25.11.2021

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I want to thank Prof. Dr. Harald Gall for giving me the opportunity of writing my thesis at the department of software evolution and architecture. I give my thanks to Dr. Pasquale Salza and Marco Edoardo Palma for supervising my thesis, and for being patient while waiting for progress updates on my part. I also want to thank my family for supporting me emotionally in this stressful time, and especially my father, for the clarifying discussions when I got stuck at some points.

Abstract

REPLs play an important part in the programming world. They have many useful features, but are lacking in user-friendliness. This thesis presents the design and implementation of a web application built around a *Python* console, aimed at improving the user experience of the console with additional features. The main addition is an interactive visualization of the command history, helping users keep an overview over what has been programmed already, letting them restore previous program states to try something else, and generating a script from the command history that can be used in other environments.

Zusammenfassung

REPLs spielen eine wichtige Rolle in der Welt der Programmierung. Sie haben viele nützliche Funktionen, aber es fehlt ihnen an Benutzerfreundlichkeit. Diese Arbeit stellt das Design und die Implementierung einer Webanwendung vor, die auf einer *Python*-Konsole basiert und darauf abzielt, die Benutzererfahrung der Konsole mit zusätzlichen Funktionen zu ergänzen. Die wichtigste Neuerung ist eine interaktive Graphi des Befehlsverlaufs, die den Benutzern hilft, einen Überblick über das bereits Programmierte zu behalten, frühere Programmzustände wiederherzustellen, um danach etwas anderes auszuprobieren, und ein Programmskript aus dem Befehlsverlauf zu erzeugen, das in anderen Umgebungen weiter verwendet werden kann.

Contents

1 Introduction **1**

2 Background **3**
2.1 Interactive Consoles . . . 3
2.1.1 REPL Definition . . . 3
2.1.2 Examples . . . 3
2.2 Python . . . 3
2.2.1 Features . . . 3

3 Related Work **5**
3.1 Educational Programming Languages . . . 5
3.1.1 Text-Based EPLs . . . 5
3.1.2 Graphical EPLs . . . 5
3.1.3 Block-Based EPLs . . . 6
3.2 Third-Party Python Consoles . . . 6
3.3 Other Tools . . . 7
3.3.1 Python Standard Library . . . 7
3.3.2 Dynamic Visualization of Data Structures with Debug Visualizer . . . 7

4 Approach **9**
4.1 Requirements . . . 9
4.2 User Stories . . . 9
4.2.1 Personas . . . 9
4.2.2 Stories . . . 10
4.3 Features . . . 10
4.3.1 Visualization of Command History . . . 10
4.3.2 Branching . . . 11
4.3.3 Editing Input And Program State . . . 11
4.3.4 Generating a Script . . . 11
4.4 Model . . . 11
4.4.1 Console . . . 11
4.4.2 Back End . . . 11
4.4.3 Front End . . . 12
4.5 Implementation . . . 13
4.5.1 Console . . . 13
4.5.2 Back End . . . 13
4.5.3 Front End . . . 15

4.5.4 WebSocket Message Protocol 19
4.6 Design Choices and Limitations 20

5 Conclusion **23**

List of Figures

4.1 Light (left) and dark (right) modes of the user interface. 16
4.2 Every type of marked node in the state visualization. 16
4.3 The editing menu of the application. 17

List of Tables

List of Listings

4.1 *Python3* command used to get the program state after every user input. 15
4.2 Summary of message format for client-to-server. 20
4.3 Summary of message format for server-to-client. 20

Chapter 1

Introduction

According to the *Merriam-Webster* Dictionary, in computing, a program is defined as "a sequence of instructions that can be inserted into a mechanism, such as a computer" [9]. The term "to program" has first been used in computing in 1942 by Mauchly, in a paper on electronic computing [38]. He used it in the sense of connecting different computing units with signal cables. The modern understanding, as defined above, seems to have stabilised only around 1951 [33].

There are different ways of programming. In most cases, programmers write their code in text files. These scripts, or programs are then executed as units. This process has three separate phases. The first step is to design a solution, the second step is to write the program code. Then the program needs to be compiled, and finally it can be executed, and the outcome validated.

Interactive programming has a different approach, where these steps happen in parallel, which is beneficial to both learners and experts. One can start programming without already knowing what the final solution is going to be. This is helpful when designing a solution or an algorithm. Additionally, any feedback is immediate, which supports learning by doing and a trial-and-error approach. However, the development environment has to support this interactive style of programming.

One application of interactive programming is called live coding. Originating in performance arts, it is also used in programming-related lectures and conference presentations [53]. It applies traditional development tools, such as monitoring file changes or automatically reloading browser pages, in an interactive fashion.

The REPL is a more traditional interactive programming environment. The acronym stands for read-eval-print-loop, which describes the coding process with such a tool. The user gives an input statement to the console, which is read by the REPL, then evaluated. The output of the statement is printed to the console, before the REPL waits for new input to repeat this process.

These consoles are useful for small tasks, experimentation with the programming language, and for learning the basics of programming.

Nevertheless, the REPL has limitations, which can be improved upon. Statements are executed linearly; there is no possibility to properly undo previous inputs, and program states cannot be saved to return to them later, and try an alternative path. In fact, there is not an easy way to get an intuitive presentation of the current program state. Even more complications occur when trying to understand and retrace how the program state evolved, since there is no insight into previous program states.

In addition, it is generally not intended to export the executed code to use it in another project. While it is technically possible to find the console history, it is not obvious and a manual process. Consoles are traditionally fully textual. To give users a better overview over the state and evolution of the program, informative visualizations could be used.

The goal of this thesis is therefore to implement an environment with the aforementioned improvements to a traditional REPL. More specifically, this tool is an interactive environment

with a visualization of the program state and command history, the ability to navigate between program states and branch off of them, and with automatic generation of a script based on the command history.

The thesis structure is the following: Chapter 2 contains background information on interactive consoles and the programming language *Python*, whose interactive mode constitutes the base for the tool. In chapter 3, we look at similar tools, and their approaches. The approach is detailed in chapter 4. We list the ideal requirements, and explain the features, then we describe the architecture and implementation, and we specify the reasoning for some design choices and the limitations of this approach. Finally, chapter 5 contains the conclusion and opportunities for future work.

Chapter 2

Background

This thesis is based on different topics, the most relevant being interactive consoles and the programming language *Python*.

2.1 Interactive Consoles

2.1.1 REPL Definition

The REPL, short for "read-eval-print-loop", is a type of programming environment. Its main characteristic is right in the name. The REPL *reads* user-given input, *evaluates* it, and *prints* the result. This process is repeated in a *loop*, as long as there is an input to read. It is also known as an interactive shell or console, or a command-line interface.

2.1.2 Examples

One of the earliest implementations of a REPL is based on *Lisp*, from 1960 [39]. *Lisp* is one of the oldest high-level programming languages, originally created as a mathematical notation. A different example is the *APL\360* terminal system from 1967, specialized on mathematics calculations [12]. Other popular consoles are *sh* and its descendants, which are used in *Unix*-based operating systems, the *R* console [32] used mostly for statistics and data analysis, and *Python*'s interactive mode [16, 18, 20].

2.2 Python

Python is an free and open-source [28] programming language. *Python* was designed by Guido van Rossum in the late 1980s [51].

The third major version, *Python3* first released in December 2008 [49]. At the time of writing (November 2021), the latest release version is 3.10.0.

2.2.1 Features

Python is a cross-platform, high-level, interpreted programming language. It is mainly object-oriented, but it also supports other programming paradigms, such as functional and procedural programming [24].

Readability

Python has a focus on readable and non-verbose code, with an official coding style guide [50]. *Python* uses indentation [19] to delimit blocks instead of parentheses or curly braces like many other languages (*C*, *Java* etc.).

Extensibility

Another important characteristic of the language is its modularity and extensibility [51]. *Python* has a large standard library [29] of modules, and there are many third-party modules available also [26].

Interactive Mode

Another feature of *Python* is the interactive interpreter [20]. It acts as a console that evaluates and executes *Python* code line by line. It has all of the features that *Python* has when writing scripts, while also, like other REPLs, printing the result of expressions.

Chapter 3

Related Work

There are many efforts in research and in the industry to create programming languages and tools to make programming more accessible to newcomers, and more fun to learn, as well as improving tools used by professional programmers for ease of use.

This chapter lists and explains some of those languages and tools, focusing on educational programming languages and third-party *Python* consoles.

3.1 Educational Programming Languages

There are different approaches to teaching the basics of programming. One way is using educational programming languages (EPL). Educational programming languages are designed to facilitate learning to program for beginners. They are mostly used to teach basic programming concepts, before learners start to learn classical programming languages with additional complexity and details.

The main types of EPLs are text-based languages, graphical programming languages, and block-based languages.

3.1.1 Text-Based EPLs

Purely textual educational programming languages are the closest to most classical programming languages. On one hand, this closeness makes it easy to switch from an EPL to a classical programming language. On the other hand, they are the more difficult EPLs to learn, since the learners have to overcome their syntax first, before being able to use them.

Many EPLs are initially based on other programming languages, such as *BASIC*, which was heavily influenced by *ALGOL* and *FORTRAN* [34].

BASIC was designed to be usable by students in non-science fields [34]. It became widely used in the 1970s and 80s thanks to the spread of home computers, many of which came with *BASIC* installed [52].

Another textual educational programming language is *A++* [36], which is based on Lambda calculus [35], a notation for mathematical application of functions to arguments [1].

3.1.2 Graphical EPLs

Another type of educational programming languages are graphical EPLs. They are different from text-based EPLs in that the program written in text produces a graphical output, typically in a two-dimensional environment.

Logo is an example of such a language. Its most popular environment lets the user control a graphics object represented by a turtle through written commands. [13] The turtle can move `forward`, turn `right`, and do only a few other simple actions. The simple interface reduces the syntax and number of commands that learners will have to memorize, lowering the entry barrier.

The immediate visual feedback and the simple commands of *Logo* and similar programming environments like *Karel* [43] make them a popular choice for teaching children about programming.

3.1.3 Block-Based EPLs

A third approach is using "building block"-like structures to build a script. These block-based languages are very different to traditional programming languages, in that users do not write code in text. Instead, they manipulate the code by drag-and-dropping different block-like elements in a certain order. These blocks represent control structures, functions, operators, and other elements used in structural programming.

Proponents of this technique argue that "seeing and pointing", i.e. direct manipulation, is a better user interface than "typing and remembering" [46].

Scratch [31], a free and open-source block-based programming environment, is aimed at children between the ages of 8 and 16 [30]. The user can control and animate sprites using blocks as described above. It is based on *Google*'s *Blockly*, a *Javascript* library for an "editor ... that represents coding concepts as interlocking blocks" [8].

There are other tools which also use *Blockly* as their base. The *Ozobot* project [42] is about programming a small robot using either markers and paper, or *OzoBlockly* [41], an editor based on *Blockly*.

A hybrid approach between block-based and textual programming is used by *mBlock* [37], a tool inspired by *Scratch* [2]. The block-based editor is similar to *Scratch*, but the user can read the textual *Python* script generated by the blocks. Additionally, there is a *Python* editor to write scripts with the same functionality in *Python* directly, using a *Python* library.

3.2 Third-Party Python Consoles

Python comes with a built-in interactive mode [18], which can be started in a terminal. On compatible shells, the console has some useful features by default [17]. Tab completion auto-completes variables and module names when pressing tab; the history is saved between sessions, and can be navigated with the arrow keys.

For more advanced features, the *Python* documentation recommends using the third-party tools *IPython* [44] or *bpython* [6].

IPython has many more features [47], some of which extend beyond the typical characteristics [48] of a REPL. It has built-in access to the *Python pdb* debugger [25], its profiler, and allows object introspection. It also supports accessing the underlying shell while running, and several "magic" commands that control the environment, i.e. *IPython* and the operating system.

While *IPython* has a feature set more similar to an IDE, *bpython* has a simpler approach [5]. Its idea is to be a console with some small but useful extra features. The "Rewind" functionality [5] is noteworthy. It allows the user to undo the last line, leading to the code up until but excluding that line being re-evaluated.

ptpython [45] is a another third-party *Python* console. It is different from *IPython* and *bpython* in that it lets the user reuse one or more lines of the previous command history without manually having to select every single line using the up arrow.

3.3 Other Tools

3.3.1 Python Standard Library

Th *Python* standard library [29] contains several modules, which are useful for a variety of tasks. Of special interest are those modules, which give access to a lower layer of the language itself, or the execution of some code. For the latter, `pdb` [25] is a debugger included in the *Python* standard library.

For the language itself, there are some modules grouped under the "Python Language Services" [27]. They make it possible to work with the *Python* language. The `ast` [22] module, for example, is useful to create and manage abstract syntax trees for *Python*.

To build custom interactive interpreters, the `code` module [23] exposes appropriate classes and functions.

3.3.2 Dynamic Visualization of Data Structures with Debug Visualizer

The *Debug Visualizer* project [10, 11] is an extension for the *Visual Studio Code* IDE [40].

This extension enables different visualizations that may be useful while programming. To get the data to visualize, the programmer can make a helper function to transform it into *JSON* that the extension uses to generate a visualization.

Chapter 4

Approach

In this chapter, we first list the requirements for the application. Then we describe the design of the application, before moving on to the implementation.

4.1 Requirements

The purpose of the application is the extension of a REPL while keeping its benefits and improving the aspects that it has drawbacks in.

The basic requirements are:

- managing a REPL in a shell
- piping user input from the application to the console
- piping the output of the console back to the application
- showing the current program state
- saving the command history
- visualizing the command history in an interactive, and dynamically updated graph
- restoring existing program states
- branching off from previous paths

4.2 User Stories

User stories are a good tool to get a better idea of what and whose needs the application is supposed to fulfill. Different types of users might have other reasons to use the tool. First, we describe these personas, then the user stories.

4.2.1 Personas

The Beginner

The Beginner does not have any, or only very little previous programming experience. They want to learn programming, and need fast feedback and support.

The Algorithm/Solution Developer

The Algorithm or Solution Developer wants to solve a problem by designing an appropriate algorithm. They need to be able to change previous parts of the code, and to see the effects that those changes have. Additionally they want to export the work done in the application into their own code base.

The Undecided

The Undecided has previous programming experience, and wants to try out different programming languages or frameworks before making a choice, ideally without having to spend a lot of time and energy installing tools on every option that they are considering.

The Tinkerer

The Tinkerer likes experimenting with code, and has fun playing around with the features.

4.2.2 Stories

- As a user, I want a summary of the current program state, so that I know what constants and variables are currently in memory.
- As a user, I want a graphical view of the history of my commands, so that I know which command led to what program state.
- As a user, I want to restore any previous program state, so that I do not have to manually reenter every command to reach the given program state.
- As a user, I want to branch off an existing program state, so that I can explore different code sequences and the resulting program states.
- As a user, I want to annotate code with my intentions, so that I can track whether I fulfill them.
- As a user, I want to export parts or all of my code as a script, so that I can use it in other development environments.

4.3 Features

The application described in this thesis is meant as a proof-of-concept for an interactive console environment that provides a visualization and navigation of program state and command history. This additional functionality is supposed to make using the console more user-friendly, better suited towards experimentation, and the resulting code easily reusable.

4.3.1 Visualization of Command History

In a console, it is not always easy to keep an overview of what you have programmed so far due to a purely textual interface and a non-trivially accessible command history.

Therefore, in this tool, the command history will be visualized as an interactive and dynamically updated graph. It allows the user to always have an overview over what they have done so far, and it is also the interface to navigate between program states.

4.3.2 Branching

Consoles typically save the command history, which is accessible in the interface going backwards chronologically line by line. So, if after working on one path for some time, the user decides to return to a previous point in the history to try something different, they have to manually re-execute the commands up to that point, going back in the history for every single command.

We address this issue with the branching functionality. Selecting a node in the history graph restores the corresponding program state. From there, the user can enter new commands, creating a new branching path in the history. The original path is saved, and any state on that path can still be restored. Both paths will obviously be represented on the command history visualization like branches of a tree.

4.3.3 Editing Input And Program State

In a traditional console, if you want to execute a sequence of commands, but with a single command changed in the middle, or different variables, you have to enter every command from that point on again manually.

To mitigate this problem, the user can edit existing commands and update, add, or remove variables for any previous program state in the tool. Like this, they do not have to start from scratch when they make a mistake.

4.3.4 Generating a Script

It is possible to access and reuse the command history of a typical console, since it is usually saved in a file. However, many users would not know that the file exists or where it is located. This makes it hard to reuse code written in a console in a different code base.

Our application makes the code written by the user reusable. The user can choose a path from the root to any node and export a script containing the sequence of commands from the beginning to that program state, with all of the edits as well.

4.4 Model

The application is modelled into three main components: a console, the back end, and the front end. The application is separated into a front end and a back end, because the program managing the console needs access to the operating system, which needs to have the REPL installed. To minimize the start-up procedure for the user, the console is run on a server, so that the user only needs to have access to the client application, and does not have to install any REPL onto their device.

4.4.1 Console

The console is an interactive console, running in a shell on a server, where it is managed by the back end.

4.4.2 Back End

The back end has access to the operating system of its host server. It is responsible for starting and stopping the console in a shell. It intercepts the standard output `stdout`, and error `stderr`

of that shell.

It also upholds a connection to the front end. Through this connection, it receives the commands to start, stop, or reset the console. It also receives the user's input, which is piped to the console's standard input `stdin`. Conversely, it sends the `stdout` and `stderr` of the console back to the front end.

4.4.3 Front End

The front end is the part of the application that the user interacts with. The main parts are the textual representation of the console, the state management, and the console history visualization.

State Management

Console History. The console history saves the data from every input piped to `stdin`, the data piped from every output to `stdout` and every error to `stderr`. The output and error data are grouped with the input that came immediately before.

Program State. The program state contains all of the constants and variables that are defined in a certain moment during the execution of a program, with their respective names, values, and types. It is collected after every input, and grouped with said input.

State Management Structure. All of this data is stored in a tree structure. A node contains one input line, any corresponding output and error, and the program state after the execution of that input. A tree structure is well suited for this data, because the branching functionality can be represented by the branches of the tree.

The root node is the program state after the console starts up. The child of a node is the next entered command and its associated data. In a linear command history, every node in the tree would thus have at most one child node. However, the tree structure allows nodes to have multiple children. This enables the branching functionality.

The State structure has some typical tree operations, but also some more specific to its meaning of a program state and command history manager.

Console Representation

While the console is running on the back end, the application represents the console to the user on the front end as well, since the user typically does not have access to the back end.

Command History Visualization

The visualization of the command history as a tree graph is an integral part of the application. The nodes represent the program states, and the edges represent the user-given inputs that lead from the parent node, or program state, to the child node's represented state. Depending on the input that leads to a state, the node is marked to add a visual indicator of program flow. For example, loops, namely `for` and `while` statements, are marked with a loop symbol. The edges are overlaid with the text of the input or the sequence of inputs that they represent.

The visualization is interactive, since selecting a node restores the corresponding program state. Additionally, the program state and the input are editable through the visualization.

4.5 Implementation

The application is implemented in a simple client-server architecture. The back-end server is a *Node.js* [14] script. In the script, the client-side web application, the front end, is also started locally. The web application's code is all in *Javascript* files referenced in the *HTML* of the web page. The communication between back and front end happens through a *WebSocket* connection. The server spawns a shell and manages a *Python3* console.

Choice of technologies. Since I have experience working with web technologies, namely the combination of *HTML*, *CSS*, and *JavaScript*, a web application makes the most sense. It also runs on most modern devices without having to port the application to different operating systems. Additionally, I also like implementing visualizations in *SVG*, which integrates well into a web page.

Node.js is a *JavaScript* runtime, therefore both the front and the back end are implemented in *JavaScript*, minimizing the number of different programming languages used.

The reason *WebSocket*s are used to connect the web application and the server-side is that it is another web technology with a simple interface, that is used for real-time bi-directional communication.

Python is a widely used programming language, used often to teach programming as well, that I know well.

4.5.1 Console

The console is the standard *Python3* interactive mode. The shell's `stdin`, `stdout`, and `stderr` are piped from and to the server script with *Node.js* streams [15]. The standard input channel receives the user's input from server-side. The standard output and error channels are piped to the server.

The *Python* console uses the output channel if the input given is an expression to be evaluated, or if a `print`-statement is given. The error channel is used, as the name suggests, if during the execution, an exception is raised and not caught, but also for the startup message and the prompts (">>> " and "... ").

4.5.2 Back End

The server manages the *Python* console and executes the messages sent from the web application through a *WebSocket* connection.

To start the server, the script needs to be run with *Node.js*. The basic command is `node index.js [PATH-TO-PYTHON]`. If the optional `[PATH-TO-PYTHON]` argument is not given, the *Python3* path is assumed to be `python3`, otherwise the given path is used to start the console.

Starting the Console. When the console is started, triggered by a message sent from the web application, all of the control variables and the streams piped from and to the console are initialized, and the client-side is sent an informational message stating whether or not spawning the shell and starting the console have been successful.

Unique ID generation. There is an incremental counter that generates a unique ID for every grouping of data, that is the input, the preceding console prompt, any output and error messages that the console prints after that input and before the next input, and finally the program state after the input's execution.

Console Prompts as Control Flow Indicators. The `stderr` stream is in a flowing state, meaning that any data chunks are consumed as soon as they are available. Since the prompts for new input, the start of the read-eval-print-loop, are piped through the error channel, they are the main indicator of control flow.

There are two types of prompts: the primary prompt ">>> " and the secondary prompt "... ". The primary prompt prompts for the next command, the secondary prompt for a continuation of the previous command.

User Input. A primary prompt means that the `stdin` stream of the console is empty, and the console is awaiting new input. At that point, a new ID is generated, and the prompt is sent to the web application with that ID. If user input is received through the connection to the front end when the console is in that state, it is pushed into the shell's `stdin`, and a message with the input and the same ID is sent back to the web application.

Collecting Output and Errors. When the console consumes the input, it will eventually print another prompt, primary or secondary. If it is secondary, so "... ", there is no other output, a new ID is generated and the new prompt is sent back to the client-side with this ID.

If the prompt is primary, a few more steps are executed. First, the output to `stdout` is collected, if there is any. It is then sent to the front end with the same ID that was used for the input. If the error stream held other data before the prompt, it is also sent back with that ID. Before restarting the loop from the start, the program state needs to be collected.

Collecting Program State. After the execution of every user-given input command, when the primary prompt is printed, the program state is queried by pushing a hard-coded *Python* command to `stdin`. It is a sequence of statements in one line, separated by semi-colons, so as to use only one iteration of the REPL per program state.

It executes the following steps:

1. Assign a copy of a dictionary map to "`___g`"[1], containing every globally defined variable with the key being its name and the value its value.
2. Assign a copy of a dictionary to "`___l`", containing every locally defined variable similarly to "`___g`".
3. Update "`___g`" with the key-value pairs of "`___l`".
4. Delete the entry for "`___g`" in the dictionary "`___g`". Delete "`___l`".
5. Import the "`json.dumps`" function, which converts a given *Python* object into *JSON*, under the name "`___dumps`".
6. Print the converted *JSON* string with...
7. the variable name, value, ...
8. and type...
9. of every item in "`___g`".
10. Delete "`___g`" and "`___dumps`".

[1]The variable names used in this command all start with a triple underscore to minimize the risk of corrupting a variable or import defined by the user.

```
'___g=globals().copy();' +
 '___l=locals().copy();' +
 '___g.update(___l);' +
 'del ___g["___g"];del ___l;' +
 'from json import dumps as ___dumps;' +
 'print(___dumps(' +
 '{x:{"name":x,"value":str(___g[x]),' +
 '"type":str(type(___g[x]))[8:-2]} ' +
 'for x in ___g},separators=(",",":")));' +
 'del ___g;del ___dumps\n'
```

Listing 4.1: *Python3* command used to get the program state after every user input.

A new primary prompt in the error stream marks that the output of this command is ready in the `stdout` stream. It is a *JSON* string of a map, where the keys are variable names, and their respective value is the name, value, and type of that variable. The program sends it to the web application through the *WebSocket* connection, with the current ID.

This time, the ID is incremented again, and attached to the prompt, is sent to the front end also. Now the console is ready to receive more user-given input, so that the cycle can start again.

Restoring Old Program States. When the user wants to restore an old program state, the message from the web application contains the sequence of commands that lead up to that state.

On the back end, the console is stopped and a new instance is started. The text data is split into an array of strings representing single lines of code, i.e. the user-given commands.

Now, the first command is pushed into `stdin`, starting the read-eval-print-loop as we described it before. When the console is ready for new input, instead of waiting for the user to enter a new command, the next line in the array is used instead. This is done until every element in the array of commands has been executed.

The program state has therefore been restored, and the server resumes the previous routine.

Stopping the Console. When the order to stop the console comes from the front end, the child process that is running the *Python* console is killed immediately.

4.5.3 Front End

The front end is the application that the user interacts with. It is a web application written in vanilla *HTML5*, styled with a *CSS* style sheet, with the functionality brought by five *JavaScript* files referenced via `<script>` tags in the *HTML*. The visualization of the command history is generated with *SVG*.

After testing with different browsers, the application seems to be compatible with most modern browsers on desktop.

User Interface. The user interface is optimized for a desktop screen. It has four sections.

In the lower middle of the view-port is the virtual representation of the *Python* console that is running on the back end. Every input to `stdin`, every output to `stdout` and `stderr` on the actual console is also printed on this virtual console. Additionally, informational messages about the status of the console are also displayed there.

The panel on the left side displays a program state at the top, usually the current one, unless the user has selected a different node on the visualization. Underneath it, there are controls to

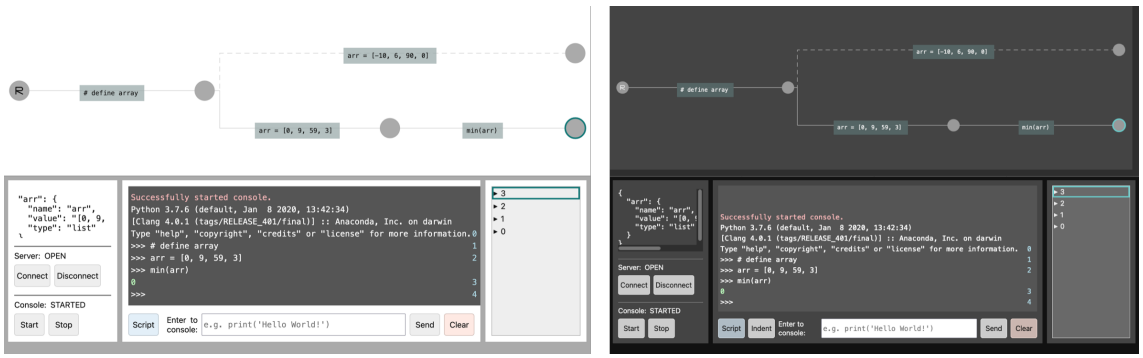

Figure 4.1: Light (left) and dark (right) modes of the user interface.

open and close the *WebSocket* connection, and buttons to start and stop the console, with the current statuses of both of those.

The right-side panel is a list of all of the program states in the currently active path. They can be related to the relevant console output by their ID. Showing the content of each program state can be toggled by activating its list entry. The active program state, and the state whose information is displayed on the left-hand side, are both highlighted with a solid and a dashed border respectively.

The largest section of the interface is the interactive graph visualization of the state tree data structure. It occupies the top half of the window. The nodes, depicted as gray circles, represent program states. If the preceding command is of a special type, they have related symbols in the middle.

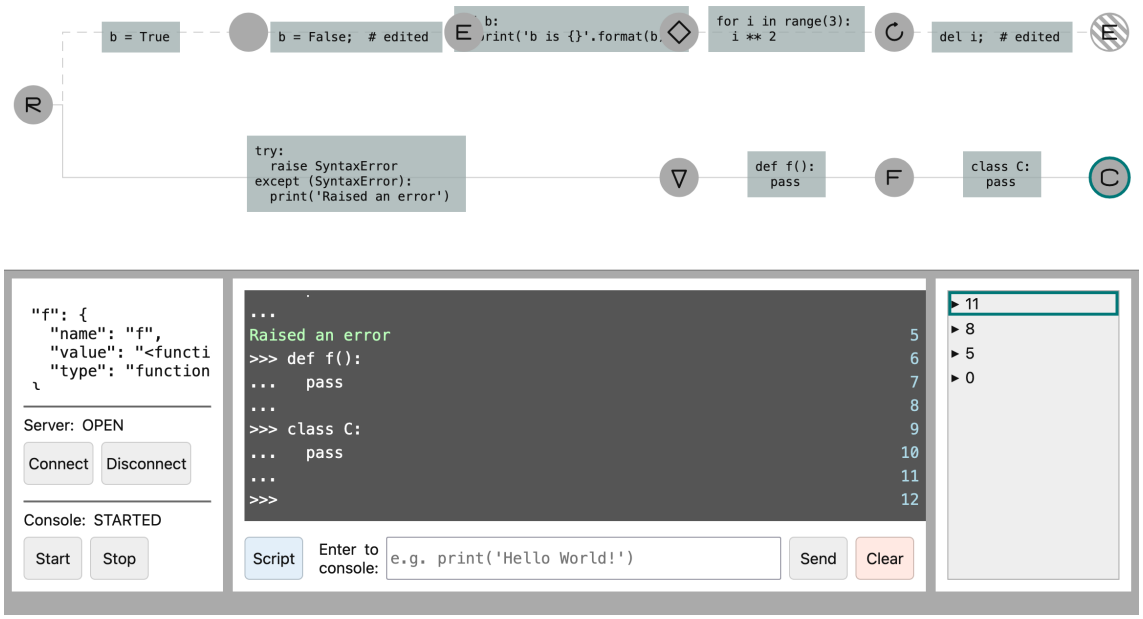

Figure 4.2: Every type of marked node in the state visualization.

The edges of the graph represent the command(s) that lead from the program state of the source node to the target node's program state. Likewise, they are depicted as lines overlaid with the text of these commands.

The nodes are interactive. Moving the cursor over a node selects it with a dashed border, and displays its program state on the left-side panel. Double-clicking a node restores that program state. The active program state, which a new input would be executed from, has its representing node circled with a solid border. A click on a node while holding the Shift-key opens up the editing menu for that state.

In the editing menu, two things can be edited: the program state, and the commands that lead from the last program state to the state to be edited. Program state is the set of all variables and constants defined at a given point in time of the execution. The user can change the values of existing variables, define new variables, and delete old variables. Alternatively, the statement leading to that program state can also be edited. The user may delete the statement entirely, add more statements, or edit the original statement.

On re-execution of an edited path, the program state is updated accordingly. If the number of statements in the edit is different from the original, this will also lead to fewer or more program state nodes between the originally edited state and its parent node. If the user has changed the program state by manipulating the variables, this change is converted into a *Python* statement, which is executed immediately after the edited state.

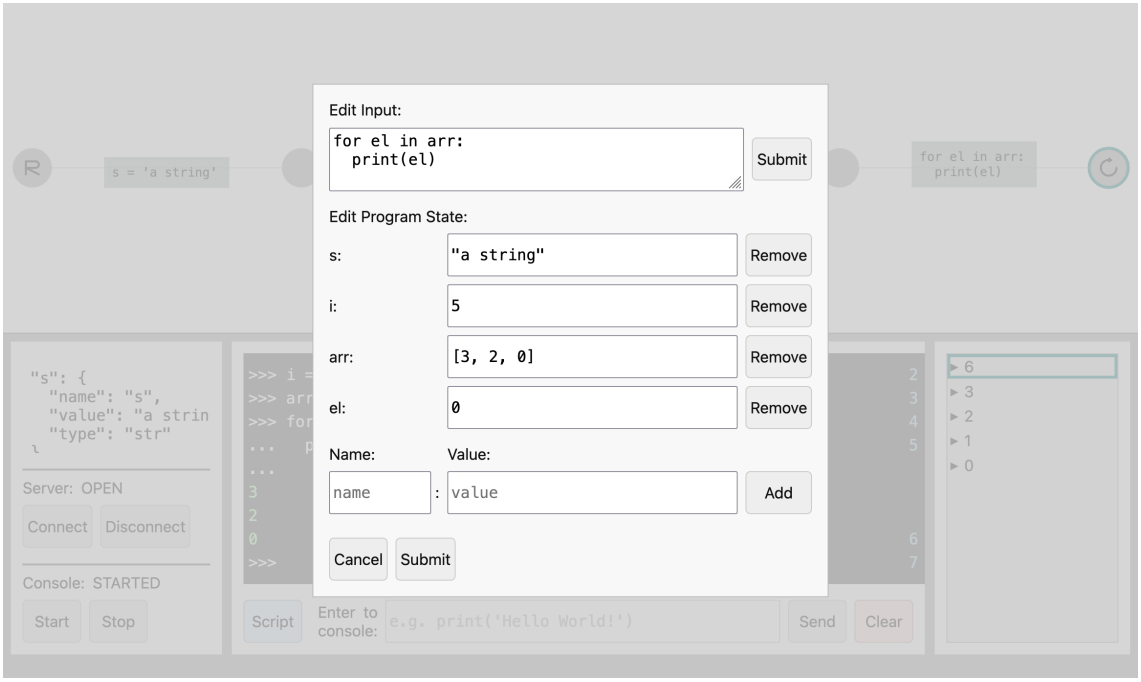

Figure 4.3: The editing menu of the application.

HTML and CSS. The *HTML* is not very complex, aside from the hand-coded modal dialog perhaps. It has basic keyboard accessibility, excluding the *SVG* visualization.

The *CSS* style sheet contains the style rules for the *HTML* document. The web application has light and dark mode, both style and color choices contained in the style sheet as well.

Scripts. The first script is an external open-source library, the other four are the result of this thesis project. The latter are loosely separated into different concerns, however there is some overlap and the later scripts also depend on the earlier ones.

D3.js. The first script is the *D3.js* library [4] for *JavaScript*. Its self-declared purpose is to "manipulat[e] documents based on data" [4]. We use it to simplify handling the dynamic updating of the virtual console and the graph visualization.

State Tree Data Structure. The second script defines the `State` data structure. It is a tree structure at its core, with additional properties and methods that are specific to its program state and console history management aspect.

WebSocket Connection and Storage. The third script deals with the *WebSocket* connection and with the management of the browser's local storage. It also initializes some DOM elements that are interconnected with those two aspects of the application.

The state of the application is saved to local storage after every change because local storage is persistent across browser sessions, as long as the user does not delete it manually. This way, if the user closes the browser after working in the application, their session will be reloaded when they open it again (provided the server is running). It is also safe to reload the page without data loss.

A custom `setStorageItem(key, value)` function ensures that if the value of an item in local storage changes, a custom DOM event is dispatched, which triggers a dynamic update of the user interface, most importantly the virtual console and the visualization.

The *WebSocket* connection can be opened – and closed – by the user by activating a button. The status of the connection is saved in local storage, so that the connection is automatically reopened on an accidental page reload.

The most complex and arguably most important part of the third script is how incoming console history and program state data is handled. If the parsed incoming message contains an ID and either of the following properties: `prompt`, `stdin`, `stdout`, `stderr` or `programState`, the State structure, as it is saved in local storage needs to be updated.

1. The algorithm checks if the message data is originally from restoring an existing program state. If that is the case, the original state is updated with the new ID and content, and the algorithm is done.
2. If the message data is new however, we check if the old tree structure is empty, i.e. only has a root node without any data. If it is, we replace it with the incoming data and return.
3. If the existing tree structure is populated, the algorithm searches for a state with an ID matching the one in the message. If this state exists, it is updated with the new data, and we stop.
4. In the final case, no state with the message ID exists. A new state with the message data is created, and appended to to the last active state as a child node.

Virtual Console and Script Generation. Since the actual *Python* console is managed from the back end, the user does not have direct access to it. Therefore, the web application is built around a virtual representation of the console and its content, which is updated dynamically with the updates coming from the back end.

Script number four is mainly focused on that virtual console, but also contains the code for generating a script from the command history, as well as other helper functions to sanitize the program state for presentation to the user.

Graph Visualization and Editing States. The last script addresses two major aspects of the application. Both the actual visualization of the state tree structure, and the editing functionality are implemented in that script.

The visualization is implemented in *SVG* using *D3.js* [4]. The visualization represents the State data structure built from the command history and program states, using *D3*'s cluster layout for trees [7]. The `update_tree(tree, size)` function updating the graph is triggered by a custom DOM event that is emitted when the underlying data structure changes, and when the window is resized. This function sorts the nodes, calculates the optimal layout for the screen size, then updates the position of the nodes and the edges accordingly, with *D3*'s "data joining" mechanism [3], binding the data to their representing *SVG* elements.

The editing functionality can be accessed by clicking on a node while holding the shift-key. This opens a modal dialog with the interface to edit the state and input, but only if the path the chosen node is in, does not contain any edited nodes that have not yet been executed by the console. (The reasoning for this behavior will be explained in section 4.6.)

The commands entered by the user between the current and the previous program state are isolated and presented in a text field. The script also collects the variables defined in the program state, excluding functions, classes and modules, since they cannot easily be re-assigned a value. The remaining variables' values are also displayed in separate text fields.

If the user chooses to submit changes to the input, the edited text is split into single commands. For each of those, a new State node is created, forming a chain of State nodes. This chain is appended to the program state before the original input, and the current program state is appended to the last of the newly created nodes, replacing the previous command history in the State data structure.

If instead, the user edits the program state directly through the variables, the submitted program state is compared to the original version. If it contains new variables, variables with changed values, or if variables have been deleted, the edited state is different from the original one. The actual change is enacted through an added *Python* command. This command assigns the new value to updated variables, defines the new variables, and deletes the removed variables, by concatenating these statements separated by semi-colons. Finally, the command is wrapped in a new State node, which is inserted after the original program state, which is equivalent to executing this generated statement list after the original program state has been reached but before any commands that come afterwards.

4.5.4 WebSocket Message Protocol

The *WebSocket* server and the client-side application are both hosted on `http://localhost:8080` as defined in the script, since this proof-of-concept tool is running locally for now. However, it will not be difficult to change that so that the server is hosted on a different machine and address than the web application.

The custom message protocol used by the *WebSocket* connection between the server and the web application is tailored to the application's needs. In both directions, it is a *JSON* string with a specific set of attributes.

Client to Server

The client-side web application sends messages to the server to tell it what to do.

The two main types are orders directly related to the console status, and orders related to the user input.

If a message is related to the user input, its type is `"command"`. The content of the message is the user's input for the console without a newline. Ideally, it would be a correct *Python* expression

or statement, but it is not if the user made a mistake.

The other type that a message from the front end can have, is "console". The content indicates what the server should do with the console, namely "start", "stop", "check", and "reset". "start" and "stop" mean to start and stop the console, respectively. "check" means that the server should reply with the console status, i.e. whether it is running or not. "reset" means that the server needs to stop and start the console again, with the "file" attribute containing the sequence of commands that the console should then execute.

```
{
    "type": ["command", "console"],
    // if type===command
    "content": "python expression/statement w/o newline",
    // if type===console
    "content": ["start", "stop", "check", "reset"]
    // if "content"==="reset"
    "file": "binary data from blob"
}
```

Listing 4.2: Summary of message format for client-to-server.

Server to Client

The message format for the server to send to client-side has more attributes than the other way around, and the main purpose is to convey information about the console and the read-eval-print-loop.

The "info" field is filled when the console has been started or stopped successfully, or if it has failed to do so.

The "status" contains the status of the console, which is either "STARTED" or "STOPPED". It is sent to the web application when the latter ordered to check the console status.

```
{
    "info": "information related to the request",
    // status of the console
    "status": ["STARTED", "STOPPED"],
    "prompt": [">>> ", "... "],
    "stdin": "command entered to stdin",
    "stdout": "data read from stdout",
    "stderr": "data read from stderr, except for prompts",
    "programState": "JSON string representing the program state",
    // if stdout, stderr, stdin, or prompt !== undefined,
    "id": "number, ID of the responsible input"
}
```

Listing 4.3: Summary of message format for server-to-client.

4.6 Design Choices and Limitations

Both the design of the application and the actual implementation are just one possible way to fulfill the requirements. The current architecture is mainly informed by what was feasible to implement in the time frame of this thesis based on my skill level and experience.

In this section we list and describe some limitations and drawbacks of the chosen design and its implementation, and what might be appropriate solutions for these issues.

Custom Interpreter. The user-written commands are directly piped to the *Python* console. The program state is also queried using *Python* commands in that console. The information that can be gathered from that data is limited. To extend the functionality and usefulness of the application, more data is needed. More information about e.g. run-time memory, the evolution of variable values, and the type of statements, could be used for better user support and assistance.

Editing States. While the application has an editing feature to update existing program states and commands, it is not technically what users might expect. Typically an editing feature changes the edited data directly, and there is the possibility to undo more recent changes.

Our application does not handle editing in this way. The main reason is that if the user edits a program state directly, the change would not reflected in the generated script, leading to confusion when the results are different from what the user expected.

To avoid that, any changes that the user makes to the program state are converted into a *Python* command that executes those changes immediately after the original program state has been reached. Not only does that ensure that a script generated from the command history actually incorporates the user's edits, it is also visible to the user in the virtual console and in the visualization, avoiding any confusion if they forget that they edited something.

There is a drawback to this method however. in *Python*, attempting to delete an name not bound to a value raises a `NameError` exception [21]. If the user edits a state such that a variable is not defined anymore, either by deleting the variable, or changing the command that defined it, and the state is in a different branch than the currently active one, the updated history is not re-executed automatically, and the program states in the State sub-tree of the edited node are not updated accordingly. Editing any of those nodes to delete the already deleted variable would lead to an error. To keep this situation from happening, if a program state has been edited, the user has to execute the new version by restoring that program state or any of its descendant nodes, before they are allowed to edit another state.

Browser Compatibility and Accessibility. As a web application, it runs in the browser. The *JavaScript* and the *CSS* for the page use some language features not compatible with old browsers or on mobile devices, but the tool runs successfully on modern desktop browsers.

Accessibility is also important. While the basic user interface of the application is keyboard-accessible, the visualization is not. This makes some of its major functionalities not usable for users who do not use a pointing device.

Chapter 5

Conclusion

Conclusion. In this thesis, we planned to implement an interactive environment, where a REPL is wrapped with a program state and command history management system, complete with an interactive visualization of the command history and the program states. First, we motivated this plan by outlining the benefits of interactive programming and REPLs specifically, and then analyzing areas of improvement.

From that, we extracted the requirements for an implementation of an application that provides the innate benefits of a REPL with features that improve on its drawbacks.

We then designed a basic client-server architecture, that connects the user of the planned application with a console running on a server. The application should present the console input and output, as well as an interactive graphical representation of the command history and the resulting program states.

Then we explained the implementation of this design in detail. The server-side *Node.js* script manages a REPL, and communicates with the front-end web application through a *WebSocket* connection.

We finished the explanation of this approach with a discussion of the design choices and the resulting implications.

Future Work. Since the application is basically a proof-of-concept, there are many opportunities to improve and extend it.

On the side of the implementation details, it has to be said that the different parts of the application, both on the web application and on the *Node.js* script, are not very well encapsulated. If the code is to be used and extended further, a refactoring may be necessary. Additionally, the accessibility features of the user interface are at the bare minimum, and will need improvement to be usable by users who depend on keyboard navigation or even a screen reader.

The script generation feature is also very basic, only letting users choose the branch of the command history for which they need the script. This feature has room for expansion, such as letting users pick and choose lines of code to include in the script, potentially in a block-based editor.

A vision of a bigger scale is using artificial intelligence to support the user more. If the user is trying to find a solution to a problem, but is stuck at one point, a digital assistant could provide hints on where the user has to look.

Finally, another opportunity for extension is to generalize the tool in the programming languages it supports. Currently, it works with *Python*'s interactive mode only, however other REPLs could benefit from such a tool as well.

Bibliography

[1] Jesse Alama and Johannes Korbmacher. The Lambda Calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.

[2] 吴贤燕. What Is mBlock 5? https://www.yuque.com/makeblock-help-center-en/mblock-5/overview, 2020. [Online; accessed 16-November-2021].

[3] Mike Bostock. Thinking with Joins. https://bost.ocks.org/mike/join/, 2012. [Online; accessed 24-November-2021].

[4] Mike Bostock. D3.js - Data-Driven Documents. https://d3js.org/, 2021. [Online; accessed 22-November-2021].

[5] bpython. About. https://bpython-interpreter.org/about.html, 2020. [Online; accessed 17-November-2021].

[6] bpython. Home. https://bpython-interpreter.org/, 2021. [Online; accessed 17-November-2021].

[7] d3. d3-hierarchy. https://github.com/d3/d3-hierarchy/tree/v3.0.1#cluster, 2021. [Online; accessed 24-November-2021].

[8] Google Developers. Blockly | Google Developers. https://developers.google.com/blockly, 2021. [Online; accessed 15-November-2021].

[9] Merriam-Webster.com Dictionary. Program | Definition of Program by Merriam-Webster. https://www.merriam-webster.com/dictionary/program, 2021. [Online; accessed 29-October-2021].

[10] Henning Dieterichs. An extension for VS Code that visualizes data during debugging. https://github.com/hediet/vscode-debug-visualizer, 2021. [Online; accessed 18-November-2021].

[11] Henning Dieterichs. Debug Visualizer. https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer, 2021. [Online; accessed 18-November-2021].

[12] A. D. Falkoff and K. E. Iverson. The APL\360 Terminal System. In *Symposium on Interactive Systems for Experimental Applied Mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium*, page 22–37, New York, NY, USA, 1967. Association for Computing Machinery.

[13] Logo Foundation. A Logo Primer. https://el.media.mit.edu/logo-foundation/what_is_logo/logo_primer.html, 2015. [Online; accessed 12-November-2021].

[14] OpenJS Foundation. Node.js. https://nodejs.org/en/, 2021. [Online; accessed 22-November-2021].

[15] OpenJS Foundation. Stream | Node.js v17.1.0 Documentation. https://nodejs.org/api/stream.html, 2021. [Online; accessed 22-November-2021].

[16] Python Software Foundation. 1. Command line and environment. https://docs.python.org/3/using/cmdline.html, 2021. [Online; accessed 17-November-2021].

[17] Python Software Foundation. 14. Interactive Input Editing and History Substitution. https://docs.python.org/3/tutorial/interactive.html, 2021. [Online; accessed 17-November-2021].

[18] Python Software Foundation. 16. Appendix. https://docs.python.org/3/tutorial/appendix.html#interactive-mode, 2021. [Online; accessed 17-November-2021].

[19] Python Software Foundation. 2. Lexical analysis. https://docs.python.org/3/reference/lexical_analysis.html#indentation, 2021. [Online: accessed 9-November-2021].

[20] Python Software Foundation. 2. Using the Python Interpreter. https://docs.python.org/3/tutorial/interpreter.html, 2021. [Online; accessed 7-November-2021].

[21] Python Software Foundation. 7. Simple statements. https://docs.python.org/3/reference/simple_stmts.html, 2021. [Online; accessed 25-November-2021].

[22] Python Software Foundation. ast – Abstract Syntax Trees. https://docs.python.org/3/library/ast.html, 2021. [Online; accessed 18-November-2021].

[23] Python Software Foundation. code – Interpreter base classes. https://docs.python.org/3/library/code.html, 2021. [Online; accessed 18-November-2021].

[24] Python Software Foundation. General Python FAQ. https://docs.python.org/3/faq/general.html, 2021. [Online; accessed 8-November-2021].

[25] Python Software Foundation. pdb – THe Python Debugger. https://docs.python.org/3/library/pdb.html, 2021. [Online; accessed 17-November-2021].

[26] Python Software Foundation. PyPI: The Python Package Index. https://pypi.org/, 2021. [Online: accessed 9-November-2021].

[27] Python Software Foundation. Python Language Services. https://docs.python.org/3/library/language.html, 2021. [Online; accessed 18-November-2021].

[28] Python Software Foundation. The Python programming language. https://github.com/python/cpython, 2021. [Online: accessed 10-November-2021].

[29] Python Software Foundation. The Python Standard Library. https://docs.python.org/3/library/index.html, 2021. [Online: accessed 9-November-2021].

[30] Scratch Foundation. About Scratch. https://scratch.mit.edu/about, 2021. [Online; accessed 14-November-2021].

[31] Scratch Foundation. Scratch - Imagine, Program, Share. https://scratch.mit.edu/, 2021. [Online; accessed 14-November-2021].

[32] The R Foundation. The R Project for Statistical Computing. https://www.r-project.org/, 2021. [Online; accessed 7-November-2021].

[33] D.A. Grier. The ENIAC, the verb "to program" and the emergence of digital computers. *IEEE Annals of the History of Computing*, 18(1):51–55, 1996.

[34] Thomas E. Kurtz. *BASIC*, page 515–537. Association for Computing Machinery, New York, NY, USA, 1978.

[35] Georg P. Loczewski. Origin. https://aplpl-intro.aplusplus.net/node17.html, 2004. [Online; accessed 12-November-2021].

[36] Georg P. Loczewski. A++: The Smallest Programming Language in the World. https://aplpl-intro.aplusplus.net/index.html, 2018. [Online; accessed 12-November-2021].

[37] Makeblock. mBlock - One-Stop Coding Platform for Teaching and Learning. hhttps://mblock.makeblock.com/en-us/, 2021. [Online; accessed 16-November-2021].

[38] John W. Mauchly. *The Use of High Speed Vacuum Tube Devices for Calculating*, pages 355–358. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[39] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, and D. Park S. Russell. *Lisp I programmer's manual*. Computation Center and Research Laboratory of Electronics (MIT), Cambridge, Massachusetts, 03 1960. [Online, accessed 12 October 2020].

[40] Microsoft. Visual Studio Code - Code editing. Redefined. https://code.visualstudio.com/, 2021. [Online; accessed 18-November-2021].

[41] Inc. Ozo EDU. OzoBlockly | OzoBot. https://ozobot.com/create/ozoblockly, 2021. [Online; accessed 15-November-2021].

[42] Inc. Ozo EDU. Ozobot | Robots to code and create with. https://ozobot.com/, 2021. [Online; accessed 15-November-2021].

[43] Richard E. Pattis. *Karel The Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, New York, 2nd edition, July 1994.

[44] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[45] prompt toolkit. A better Python REPL. https://github.com/prompt-toolkit/ptpython, 2021. [Online; accessed 18-November-2021].

[46] David Canfield Smith, Allen Cypher, and Jim Spohrer. Kidsim: Programming agents without a programming language. *Commun. ACM*, 37(7):54–67, July 1994.

[47] The IPython Development Team. IPython Documentation. https://ipython.readthedocs.io/en/stable/index.html, 2021. [Online; accessed 17-November-2021].

[48] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. A principled approach to repl interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, page 84–100, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Guido van Rossum. A Brief Timeline of Python. `http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html`, 2009. [Online; accessed 8-November-2021].

[50] Guido van Rossum, Barry Warsaw, and Nick Coghlan. PEP 8 – Style Guide for Python Code. `https://www.python.org/dev/peps/pep-0008/`, 2001. [Online: accessed 9-November-2021].

[51] Bill Venners. The Making of Python. A Conversation with Guido van Rossum, Part I. `https://www.artima.com/articles/the-making-of-python`, 2003. [Online; accessed 8-November-2021].

[52] Wikipedia contributors. BASIC — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=BASIC&oldid=1047128882`, 2021. [Online; accessed 12-November-2021].

[53] Wikipedia contributors. Live coding — Wikipedia, the free encyclopedia, 2021. [Online; accessed 25-November-2021].