

A LIVE PROGRAMMING ENVIRONMENT FOR PYTHON

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Noah Negrey

June 2017

© 2017
Noah Negrey
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: A Live Programming Environment for
Python

AUTHOR: Noah Negrey

DATE SUBMITTED: June 2017

COMMITTEE CHAIR: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Theresa Migler, Ph.D.
Lecturer of Computer Science

COMMITTEE MEMBER: John Seng, Ph.D.
Professor of Computer Science

ABSTRACT

A Live Programming Environment for Python

Noah Negrey

Learning how to program can be quite difficult. Beginners face two obstacles to learning to program: developing a mental model of the steps of computation and learning the syntax of a programming language. Because programmers are prone to make mistakes that introduce errors into their code, it is important that they understand what each line of code is doing in their program. I propose that a system be built in which writing code and running code happen simultaneously. This system shows the results of code to the user immediately after it is typed; this type of system is called a live programming environment. The environment's features allow the programmer to highlight individual lines of code to see their result. The programmer can walk through loops and explore the details of a function call. The programmer can unit test a function or class. The environment displays variables declared in the code and their values and allows the programmer to create a visualization for class objects. The environment appears to be beneficial for beginners based on the results of two surveys with a total of 27 participants. The results of this thesis show that live programming environments should be developed to help the beginner learn to program.

ACKNOWLEDGMENTS

Thanks to:

- My Family, for their continued support and love.
- Dr. Keen, for his indispensable wisdom, guidance, and input
- The entire Computer Science Department at Cal Poly
- The Cal Poly Students, for participating in the surveys

TABLE OF CONTENTS

	Page
LIST OF FIGURES	ix
CHAPTER	
1 Introduction	1
1.1 Problems Beginners Face	2
1.2 General Solution	4
1.3 Paper Overview	5
2 Related Work	6
2.1 Bret Victor	6
2.1.1 Up and Down the Ladder of Abstraction	7
2.1.2 Learnable Programming	8
2.2 Sean McDirmid	9
2.2.1 Usable Live Programming	9
2.2.2 A Live Programming Experience	10
2.3 Python Tutor	12
2.4 PyCharm	14
2.5 Light Table	17
3 The Environment	20
3.1 Overall Architecture	20
3.2 GUI	20
3.2.1 Menu	22
3.2.2 Source Code Box	22
3.2.2.1 Line Highlights	23
3.2.2.2 Loops	24
3.2.2.3 Functions	25
3.2.2.4 Objects	28
3.2.2.5 Error Handling	29
3.2.3 Variables Box	32
3.2.4 Objects Box	32

3.2.5	Input Box	34
3.2.6	Output Box	36
4	Implementation	37
4.1	Walking the Abstract Syntax Tree	37
4.2	Running the Programmer's Code	40
4.3	Displaying the Results	44
5	Validation	46
5.1	Survey Overview	46
5.1.1	Student Survey	47
5.2	Proctored Survey Results	48
5.2.1	Overall	48
5.2.2	Source Code Box	49
5.2.2.1	Clickable Loop Comments	51
5.2.2.2	Clickable Function Comments	51
5.2.3	Variables Box	51
5.2.4	Changing Program State	53
5.2.5	Input Box	53
5.2.6	Output Box	55
5.2.7	Final Comments	57
5.3	Self Administered Survey Results	58
5.3.1	Overall	58
5.3.2	Source Code Box	60
5.3.2.1	Clickable Loop Comments	61
5.3.2.2	Clickable Function Comments	62
5.3.3	Variables Box	62
5.3.4	Changing Program State	63
5.3.5	Input Box	64
5.3.6	Output Box	65
5.3.7	Final Comments	67
5.4	Results	68
6	Future Work	69
6.1	Custom Python Interpreter	69

6.2	Multiple File Support	69
6.3	Expression Granularity	69
6.4	Recursion Support	70
6.5	Survey Suggestions	70
7	Conclusion	71
	BIBLIOGRAPHY	72
APPENDICES		
A	Informed Consent Form	75
B	Environment Overview	78
C	Survey Setup	81
D	Program: Loops, Functions, Input, & Output	83
E	Program Survey: Loops, Functions, Input, & Output Survey	92

LIST OF FIGURES

Figure	Page
2.1 PythonTutor with a simple program	13
2.2 PyCharm while debugging a simple program	16
2.3 Light Table’s Inline Evaluation	18
2.4 Light Table’s Expression Evaluation	19
3.1 Live Program Environment Breakdown	21
3.2 Basic Line Highlight	23
3.3 Advanced Line Highlight	24
3.4 Advanced Line Highlight with Multiple Function Calls	25
3.5 Clickable Loop Iterations	26
3.6 Clickable Functions	27
3.7 Basic Class Test	29
3.8 Clickable Classes/Objects	30
3.9 Compiler Error	31
3.10 Compiler Error	31
3.11 Variables Box	32
3.12 Objects Graph	34
3.13 Object Information	35
3.14 Input Correction	36
4.1 Simple AST Example	39
4.2 Variable Correction	44
5.1 Survey Code	46
5.2 Proctored: Overall Responses	49
5.3 Proctored: Source Code Box Responses	51
5.4 Proctored: Variables Box Responses	52
5.5 Proctored: Sliders Responses	54
5.6 Proctored: Input Box Responses	55

5.7	Proctored: Output Box Responses	56
5.8	Online: Overall Responses	59
5.9	Online: Source Code Box Responses	61
5.10	Online: Variables Box Responses	63
5.11	Online: Sliders Responses	65
5.12	Online: Input Box Responses	66
5.13	Online: Output Box Responses	67

Chapter 1

INTRODUCTION

Learning how to program can be quite difficult. Beginners face two obstacles to learning to program: developing a mental model of the steps of computation and learning the syntax of a programming language. The reason these are such a challenge to learn is that they are intertwined. Learning the syntax of a programming language or how to write code that is understood by the computer also means understanding how the computer steps through and executes code. Although writing a program involves explicitly working with the syntax, it implicitly requires working with the model of computation for that syntax. To develop a mental model of the steps of computation, a beginner has to adapt their thought process to understand what steps a computer takes to execute a program. Deficiencies in a beginner's mental model are why some of the main challenges are understanding what each line of code is doing, finding where bugs occur in their code, and even the effect of a bug in their code.

Finding and fixing bugs is known as debugging. The debugging process consists of first running the code's test cases to see if the results are correct. If the results are not correct, changes must be made to the code before it is run again. After running the code again, the beginner must analyze the results to see if they are now accurate. The process is then repeated until the correct results are achieved. The debugging process of changing, running, and observing can be very slow and tedious for a beginner. Because programmers are prone to make mistakes that introduce errors into their code, it is important that they understand what each line of code is doing in their program.

I propose that a system be built in which writing code and running code happen simultaneously. This system allows the programmer to see the results of their code as

they write it, meaning that as the programmer types their code into the system, the system takes that code and executes it for the user. The results of that code are then shown to the user immediately after it is typed; this type of system is called a live programming environment. A core aspect of this system is that it displays information about each line of code that is executed by the computer so that beginners can have a better understanding of what each line of code is doing.

1.1 Problems Beginners Face

Programming can be expressed as a three step process. In step 1, the programmer has to analyze a problem and come up with a solution. In step 2, the programmer writes the code. In step 3, the programmer tests the code with varying inputs or commands while the program is running. Step 1 happens in a variety of forms depending on the size of the problem the programmer is trying to tackle. For simple problems, the programmer can retain an entire solution in their train of thought. As these problems increase in size, the programmer may have to develop an algorithm and use different data structures in their solution. Larger problems even require a design specification document that lays out all the steps required for the solution. However, for a programmer that is new to programming, it is often hard to determine what the solution to a problem even is, which is why they tend to start writing and running code before planning out their solution. This is where steps 2 and 3 come in: writing the code and running the program. Since beginners often gloss over step 1 (or work from instructor given guidelines), they spend their time switching between steps 2 and 3, because they don't fully understand what each line of code is doing. Therefore, they must continually make modifications to their code and run it again with different inputs to test that the code is running correctly.

Immediately writing and running code for the results forces the beginner to con-

stantly switch between two different trains of thoughts: writing the code (the syntax) and debugging the code after it is run (the steps of computation). However, even if the results are right or wrong, the programmer does not learn what their code is doing because the computer acts as a black box when running the program. Instead, they just see the incorrect results and are forced to solve the problem without any help from the computer. The incorrect results can occur from bugs that have propagated throughout the code. The computer doesn't offer any help because it is not designed to do anything other than run the program and give back the results. However, the environment developed in this thesis analyzes the program and provides information about the program.

There are many ways to debug a program. Inserting print statements is typically the first step in debugging for beginners because it is easy and it allows them to see specific parts of their code's execution, such as variable values or the results of an expression, in the output after running their program. It is almost the same as showing the state of a program while it is executing. An advanced alternative to print statements are debuggers that allow the programmer to enter step 3 (running the code) in a controlled manner, where they get to control the iteration of the program and they can evaluate different aspects of their code at a specific step. Debuggers are a challenge to beginners because debuggers are yet another thing to learn and they may not know what to look for in their code. Debuggers require the programmer to specify which variables or expressions should be evaluated. The challenge is that the programmer still needs to know what should be evaluated to discover the error in the code. The environment developed in this paper makes this process easier, by allowing the programmer to scan each line of code to see its results. Scanning each line of code allows the programmer to run the code in a controlled manner and easily evaluate each line to find where an error occurred.

Beginning programmers will benefit from a system that will help them understand

what each line of code is doing and that will provide that information immediately. Hancock [3] outlines what that system looks like in the following analogy: programmers develop programs similar to how an archer shoots arrows. They write their code, run it, and hope that they did not make a mistake; they either hit the bullseye or they miss the mark. However, if programmers had a live programming environment, it would allow them to see the results of their work as they typed it. Such a system is similar to a fire hose, where as you start aiming the water at the target you see the entire solution as you work to reach your goal. The water stream from the fire hose is similar to a live programming environment that has all the information about the running program and displays information about each line of code.

1.2 General Solution

I developed a Live Programming Environment for Python that is aimed at introductory courses and anyone new to programming. The goal of the environment is to allow beginners to explicitly explore the steps of computation by interacting with the environment as it executes. To achieve these goals, the environment executes the programmer's code as the programmer types it, gathers information about the executed code, and then displays that information back to the user. The environment allows a programmer to view the result of each line of code at the time it was executed, which is similar to putting a print statement after each line of code. The environment also acts as a debugger that allows the programmer to control which line of code is executed and traverse both backward and forward in time. The environment's design allows the programmer to experiment with computations by changing the input values and seeing how they affect the results of the program. The features of this Live Programming Environment for Python are intended to assist those learning how to program.

1.3 Paper Overview

Chapter 2 discusses the related work that sparked the idea for this environment and some different environments that achieve some of the same goals as the environment in this paper. Chapter 3 discusses the environment's implementation touching on the front facing application the programmer interacts with and how the environment gathers and presents the information to the programmer. Chapter 4 covers how the system was evaluated and the results of that evaluation. Chapter 5 discusses future work that would benefit the environment. Chapter 6 provides a summary of and concluding remarks about this work.

Chapter 2

RELATED WORK

Bret Victor wrote two essays that focus on making programming accessible and are an inspiration for the environment developed in this thesis. There are also three environments, Python Tutor, PyCharm, and Light Table, that offer features that are similar to the environment in this paper and unique to themselves. However, the environment in this paper adds some new features that are not found in the related environments. Details of each related project are given in the following sections.

2.1 Bret Victor

Bret Victor presents the basis for this thesis in two of his essays. “Up and Down the Ladder of Abstraction” [18], “a systematic approach to interactive visualization” and “Learnable Programming” [19], “designing a programming system for understanding programs.” In [18], the ladder of abstraction presents an idea that is similar to the analogy by Hancock [3] about the fire hose, where the user can view the entire state of a program at different levels of granularity. In [19], various data visualizations of the information to show for each line of code being executed are considered. Bret Victor, a member of the Human Advancement Research Community (which “aims to allow all humans to see further and understand more deeply” [4]), sets a goal to “revolutionize how people learn, understand, and create. Give scientists the tools to diagnose and cure the world’s ill, and artists the tools to create and share beauty in ways currently impossible” [17]. The two essays that I have chosen as the basis for this thesis explore and observe how programming can be revolutionized in new ways to make programming accessible. A short overview of the papers and how they are

relevant to this work is presented below.

2.1.1 Up and Down the Ladder of Abstraction

“Up and Down the Ladder of Abstraction” [18] presents the idea of a ladder that represents the execution of a program, including the source code and its results. Each rung on the ladder can change how much control a programmer has over a program and each rung can also change the granularity of the results for the program.

At the lowest rung, the flow of the current code for one run of the program is seen. This is the simplest form of how a programmer begins to program: writing the code, running the program, and viewing the results.

As the programmer steps up to the second rung on the ladder, the programmer’s level of control increases and they can control time, walking both forwards and backwards through the code. This rung on the ladder is similar to a debugger, which lets a programmer step through the code to see what each line of code is doing. However, a typical debugger only lets a programmer step forward and not backward.

As the programmer steps up onto the third rung of the ladder they can start manipulating parts of their code and see the effects in real time. In a debugger this would be similar to a programmer changing the value of a variable mid-execution. However, if the programmer were also allowed to go backward and forward in time while changing variables, then they have more control over their program. At this rung on the ladder, the programmer is first introduced to the idea of a live programming environment, in which changes made by the programmer immediately affect the results of the program.

At the fourth rung on the ladder, the abstraction grows even more, now the programmer can see the complete flow of the program. This means that the programmer can see how immediate changes affect the results and the values at each line of code

being executed. As they make changes to their code, they can see those changes propagate throughout the entire program and not just the results.

The last rung on the ladder of abstraction is the ability to see the output of the program for all possible inputs. At this level, the results are not seen in quite the same way as before, as that would create too much output to sift through. Rather at this level, the programmer would see charts and graphs with information about their program based on the input.

On the ladder of abstraction the programmer varies their control over the granularity at which they view their program by moving up or down the ladder. They can show just the output of the program or the details of execution for each line of code. This thesis will be using all but the last rung on the Ladder of Abstraction to help the programmer. At this fourth rung, the programmer uses a live programming environment that allows them to see the results for each line of code and walk both forward/backward through their code.

2.1.2 Learnable Programming

“Learnable Programming” [19] tries to present code as an interactive environment that is tangible to the programmer. The environment should help the programmer understand their code, without forcing the programmer to learn the new environment. Instead of providing just the ingredients and final results, or the code and its output, the environment should actually show the steps of execution and allow the user to interact within that environment. An environment like this should show the state of a program to the programmer.

For each line of code, information should be displayed about that line, dependent upon which rung of the ladder the programmer is on. The information can be as simple as showing the results for a line of code or going into the details of executing

each expression in that line. The information then has to be displayed back to the programmer in a meaningful way, to help them see what the computer is doing while running their program. The environment also needs to react to the programmer and their code by not only running it for the user, but also creating a visualization of their program. This thesis focuses on making each line of code understandable to the programmer by displaying information along with the flow of execution. This thesis also tries to react to the programmer by creating a visualization of data structures that the programmer creates, such as tree structures.

2.2 Sean McDirmid

Sean McDirmid has been investigating live programming environments for several years. One of his papers, “Usable Live Programming” [10], is related to this thesis and one of his workshop videos, “A Live Programming Experience” [9], presents a live programming environment that implements many of the features discussed by Bret Victor. Following is an overview of McDirmid’s work.

2.2.1 Usable Live Programming

“Usable Live Programming” [10] defines live programming as editing and debugging code at the same time. However, McDirmid notes that live feedback, which shows how the program executes, is required to improve a live environment. The paper introduces YinYang, a live programming system, that aims to provide the missing feedback through probing and tracing. Probing shows the details of how code executes. Tracing overviews the program through print-like output.

Probing an expression is a manual process that requires the placement of an @ symbol before the expression. The @ symbols tells YinYang to show the value of that expression to the programmer.

Tracing is a feature for programmers to manually add print-like statements into the code. Traces use a separate window that is meant solely for programmers, however it acts the same as traditional print statements. The trace window allows the programmer to click on the output and jump to that point in code.

The combination of probing and tracing provide a powerful set of tools to programmers, but beginners may have trouble understanding how to properly use them because they won't know what to probe or trace. Although YinYang is able to show how a program executes, it is left to the programmer to uncover those intricacies. The environment in this thesis uses features that are similar to probing and tracing, but makes information available to the programmer by default.

2.2.2 A Live Programming Experience

A Live Programming Experience [9], an online video, shows a demonstration of A Programming eXperience (APX). APX is a custom live programming environment environment that has a rich editor with live feedback that is not available for experimentation. APX is a demonstration of features that would be included in a live programming environment and it uses many of the ideas presented by Bret Victor. APX is a step toward making programming more accessible. In McDirmid's workshop presentation, he shows examples that utilize visualization by drawing shapes onto the screen, similar to Processing [1]. APX uses Bret Victor's Ladder of Abstraction to help the programmer visualize their program. The features of APX are discussed below.

APX supports code completion, where each shape or piece of data in APX has a set of default values to help the programmer. Since each shape is defined, APX supports code completion for the different facets of each shape, such as size or color. APX supports probing to evaluate the value of expressions. To view those expressions

the programmer hits ‘ctrl-p’. APX supports tracing via a ‘log’ command that prints out the value of the logged variable.

APX introduces scrubbing, which allows a programmer to select the value of a variable and move the mouse (while selected) to change its value. For example, in the assignment $x=100$, if the programmer selects the value 100, they are able to move their mouse up or down to change the value of 100 to 50 (down) or 150 (up). Scrubbing works on advanced data types such as position points that contain an (x,y) coordinate, because a position is also part of APX’s language. The programmer can select the position point in their code and move the mouse to change the values of ‘x’ and ‘y’, as if they were on an (x,y) coordinate system and see the shape move on the screen. Scrubbing allows the programmer to select a variable and assign it to several other variables. For instance, if there were variables: $x=5$, $y=5$, and $z=5$. The programmer could create a variable: $a=5$. Then select and drag ‘a’s 5 value to ‘x’s, ‘y’s, and ‘z’s 5 values and the environment would change the code to this: $a=5$, $x=a$, $y=a$, and $z=a$. Then the programmer can quickly change all the values, by just adjusting the value of a .

APX can run animations since it draws shapes on the screen. This is done through a special APX function and it allows the programmer to walk through each step of the animation. APX also shows a strobing of the animation on the screen for the next 200 steps. A strobing shows the entire animation on the screen similar to Hancock’s water hose analogy [3].

APX allows the programmer to select objects in the displayed window and edit them and those changes will be made to the source code. For example, an object can be dragged to a different point on the screen or the programmer can change the size of the object. The changes can only occur on arguments to the object. The programmer can create shapes from the display window without having to write any

code, because they have default values. Multiple shapes can be combined to create one shape.

APX creates a compelling example of the features that can be added to a live programming environment, but many of the features are focused on visual demos, whereas beginners that are learning to program rarely start with such visual programs. APX also uses a custom language. McDirmid notes that APX works well for demonstrations with 2D shapes, but is difficult to scale to actual programs. McDirmid ends the video with a question left by Bret Victor about the many features in APX, “these are design requirements for a good environment. How do we change programming to fit?” Although, APX is a sophisticated demo, it is not well-suited to standard programming practices and therefore not yet something beginners should first encounter when learning to program. The environment developed in this thesis uses Python, so that it is relevant to beginners and uses similar features to McDirmid’s probes/traces.

2.3 Python Tutor

Python Tutor [2], created by Philip Guo, is a tool that helps beginners understand what each line of code is doing in their program. At the time of writing, Python Tutor introduced an experimental live programming environment. The three main features that it offers are immediate code execution, the ability to control time by allowing the programmer to step forward/backward through the code, and the contents of stack frames. The stack frame has information about a specific scope, it holds information for variables and their current values, function pointers, and object pointers. Python Tutor, as seen in Figure 2.1, allows the programmer to quickly see the execution of their code and get a visualization of how it is arranged in a stack frame for the specific scope.

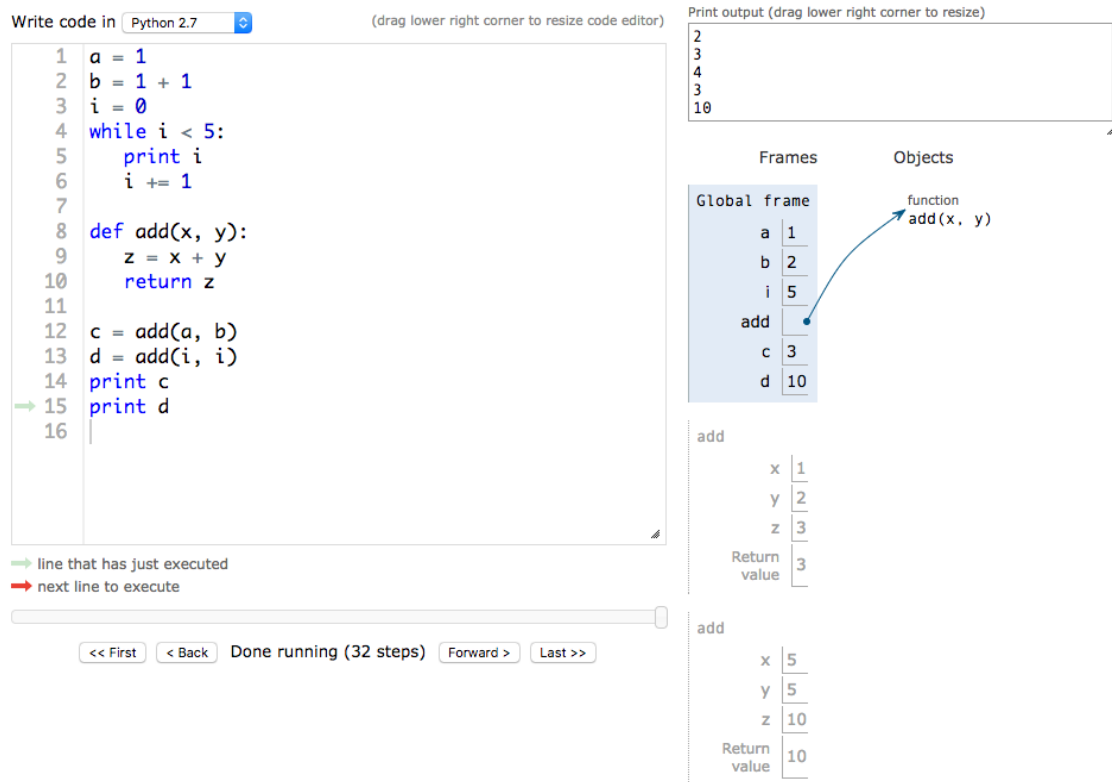


Figure 2.1: PythonTutor with a simple program

Python Tutor's advantage is the ease with which the programmer can control time and see information about the results of each line. The programmer is able to control which line of code is being shown/executed, because Python Tutor explicitly tells the programmer which line of code just executed and which line it is about to execute. Python Tutor also creates a visualization of the stack frames for the code being executed and it will show the stack frames for functions and objects. Python Tutor allows the programmer to show or hide previous stack frames, such as those for previous functions calls. For example, if you call a function, it has a stack frame, but once the function call is done, that stack frame goes away as you step through the code, but you can also have it show all previous stack frames to show all the information at once.

Although Python Tutor is really good at showing information while walking

through the code, it is stuck on the third rung on the ladder of abstraction. Python Tutor does not reach the fourth rung on the ladder of abstraction requiring abstracting away some of the information, such as the results for each line of code for the entire program. For instance, to see the result for each line of code, the programmer has to go back in time to see those values in the stack frame. It would be helpful if the programmer could see what the result is for each line of code, without having to change where they are in time when looking at a program's output. This thesis fills the gap by associating each line of code with its resulting value.

Python Tutor is an amazing program for creating a visualization of the stack frame, but that information may still seem too advanced for a beginner, who wants to learn the programming language first. This is why this thesis does not show the stack frame to the programmer, but rather lines up the code and its output.

2.4 PyCharm

PyCharm 2017.1.1 (build 171.4163.6) [6] is a sophisticated Integrated Development Environment (IDE) that provides a fully developed debugger for Python. An IDE is a graphical system that is capable of performing steps 2 and 3 for programming: writing the code and testing/running the program. IDEs typically have several features that make them appealing to beginners: syntax highlighting, auto-complete while writing code, and debuggers that make information about a running program more accessible. PyCharm provides a great workspace for Python projects, because it organizes project files and provides contextual prompts to the programmer while they are typing their code.

PyCharm includes autocompletion, which tries to predict variable names, function calls, and library imports while the programmer types, and notes what information is needed for functions calls. This information is extremely helpful for programmers

as it reduces some of the burden of having to remember all the various variables, functions, and third-party libraries that can be in a large project. PyCharm also detects and highlights compilation errors in the programmer's code and tries to offer solutions to those problems, this can solve some early headaches for the programmer.

PyCharm's debugger, as seen in Figure 2.2, provides an amazing amount of information about the programmer's code execution. The debugger allows the programmer to set breakpoints in the code. A breakpoint tells the debugger when the programmer wants to run their program in a controlled manner. The debugger then allows the programmer to control the flow of execution for the program, such as stepping over a line, stepping to the next executed line, stepping out of a function, or jumping to the cursor's location. The debugger also keeps track of the stack frames showing only the most up-to-date frames based on where the code is currently being executed, meaning that the programmer cannot step backwards in time. By selecting different stack frames, the debugger displays the variables and their values in those frames.

While a programmer steps through the code, PyCharm's debugger displays the result of that line next to the source code. The debugger also allows the programmer to select a line or part of a line of code that has executed to see that statement's or expression's result in context to the program. This means that the debugger is showing the results for different parts of one line of code that are executed by the computer. For example, take the line of code below.

```
1 a = (1 + 1) + (2 + 2) .
```

The line results in the variable 'a' being assigned 6. However, if the programmer highlights over (2 + 2), then the debugger shows just that expression's result, which is 4. Since a computer has to break down complex lines of code into smaller parts. The granularity in detail is left open to the programmer to decide how much information they wish to get from the debugger.

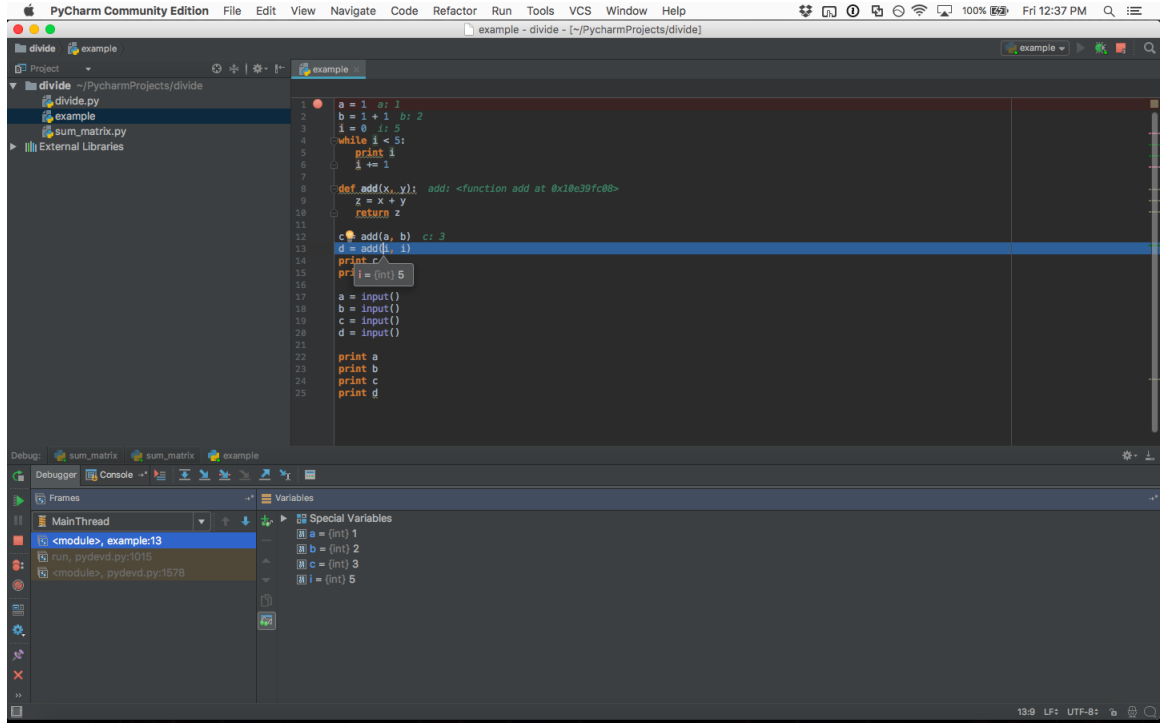


Figure 2.2: PyCharm while debugging a simple program

The significant drawback to PyCharm is that it still requires the programmer to step away from writing code to enter the debugging state. PyCharm is not a live environment in which the programmer can quickly understand each line of code or make adjustments during debugging. PyCharm is a traditional IDE, which is why this thesis seeks to amend the gap left by PyCharm. The environment in this thesis acts as that live programming environment that makes the information about each line of code easily accessible to the programmer, by directly displaying that information back to the programmer. One other problem with PyCharm is that it can appear daunting to beginners because of the multitude of features that are targeted at experienced developers. Although the debugger has a wealth of information, getting to that information can be overwhelming. To solve this issue, the environment in this paper makes information accessible by allowing the programmer to highlight over the source code to get the results without having to enter a debugging state.

2.5 Light Table

Light Table (0.8.1) [7] is a new editor currently under development that aims to streamline the code writing process. Light Table is designed to be customizable by the programmer. Light Table's features related to this thesis are quick code execution, inline evaluation, and expression evaluation. However, due to the customizable nature of Light Table, it has a steep learning curve: it is a hot-key-based program and its different features need to be activated manually.

The quick code execution feature is a hot-key-based feature that runs the code inside the editor. Light Table's code execution is capable of running a single line of code or the entire program based on the programmer's selection. This feature allows the programmer to maintain the mindset of writing code and ensuring the code runs correctly.

The inline evaluation feature shows the results of each line in the editing window. One drawback for Python, at present, in Light Table is that statements only receive a check mark, as seen in Figure 2.3. For a beginner, this provides no useful information about how each line of code is executed. The only line of code that provides useful results is a call to a function that does not use its return value.

The expression evaluation feature, correctly shows the results for some lines, but is a manual step. The programmer has to have knowledge of which expressions to select, otherwise Light Table shows an error for invalid selections. Figure 2.4 shows how some information can be extracted from a program after it is explicitly specified. However, all the steps required to show this information require knowledge of how the program works to properly select the expressions. A beginner would have a difficult time using this feature because they have to know exactly which expression to select. The editor should make expression evaluation accessible to beginners that

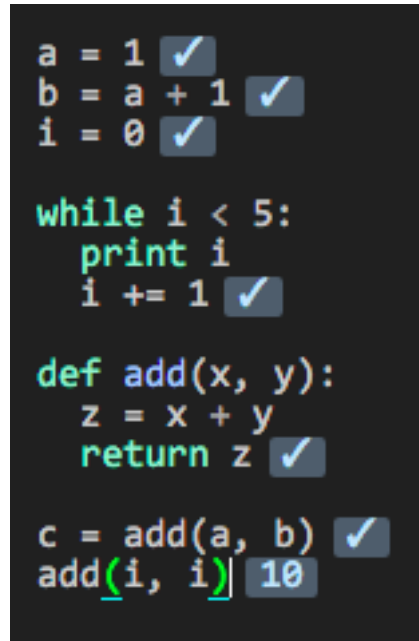


Figure 2.3: Light Table’s Inline Evaluation

don’t yet know the intricacies of a language. The environment developed in this thesis automatically provides the results to each line of code.

Although Light Table provides flexibility to a programmer, it is not an easy editor to use: hot keys must be learned; windows only appear when selected; the console output is not cleared for each run of the program (a beginner could be confused as to which output is correct); and, at present, Light Table does not support user input. Overall, Light Table is not targeted at beginners learning to program.

```

a = 1 1 ✓
b = a + 1 2 ✓
i = 0 0 ✓

while i < 5: False
    print i 4
    i += 1 ✓

def add(x, y):
    z = x + y 10
    return z 10 ✓
|
c = add(a, b) 3 ✓
add(1, 1) 10 10

```

Figure 2.4: Light Table's Expression Evaluation

Chapter 3

THE ENVIRONMENT

Bret Victor’s essays [18, 19] are where this thesis is focused, showing off the inner workings of what the computer executes for the user and how those results can be easily visualized on the screen in a live programming environment. To accomplish these tasks, one must first select the programming language to target for these features. Since the introductory courses at Cal Poly use Python, the environment targets Python. The environment is also written in Python and to make the environment accessible to a beginner, the environment uses a Graphical User Interface (GUI) that is built using Tkinter [16] in Python, a flexible library for building a GUI.

3.1 Overall Architecture

The overall architecture of the environment is broken into two systems, the front facing GUI that the programmer interacts with and the backend, discussed in Chapter 4, that runs the programmer’s code and makes the information accessible to the GUI/Programmer. The GUI drives the environment since it responds to the programmer. The GUI application constantly checks for changes in the programmer’s source code and when changes are detected, the source code is sent to the backend, which then runs the source code. The GUI also responds to actions performed by the programmer as seen in the next section.

3.2 GUI

A GUI is an important aspect in making the environment accessible to those learning to program. To create the GUI, the environment uses Tkinter for Python

[16]. The environment's GUI is broken down into six main parts: menu, source code box, variables box, objects box, input box, and output box. The programmer interacts with these six elements to write and inspect a program. The breakdown of these elements, as seen in Figure 3.1, is meant to distinguish the different pieces of information available to the programmer. The Menu field, in brown, allows the user to step forward/backward through their code. The Source Code box, in red, is where the user types their code and views the results for each line of code. The Variables box, in green, keeps track of variables and their values for each scope. The Objects box, in blue, creates a graphical representation for classes/objects that the programmer creates. The Input & Output boxes, in purple, handle any input and output that is done in the program. A breakdown of each element and its supported features can be found below.

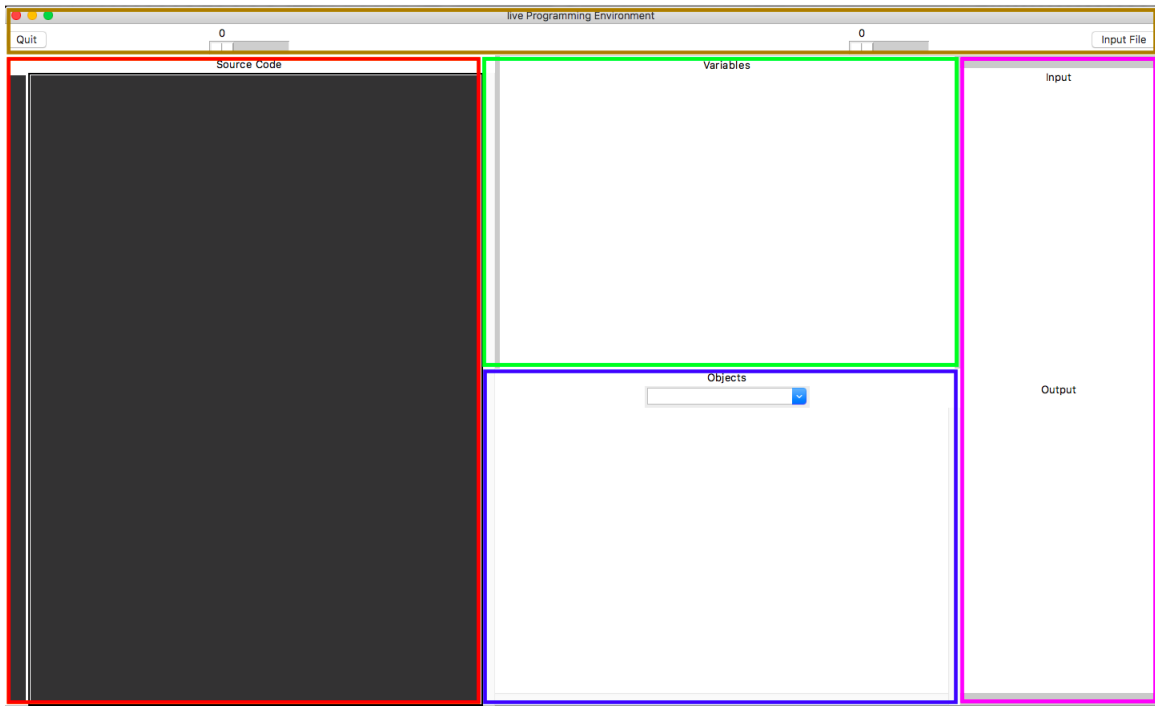


Figure 3.1: Live Program Environment Breakdown

3.2.1 Menu

Inside the Menu field, there are four elements: a ‘Quit’ button, two slider elements, and a ‘Input File’ button.

The ‘Quit’ button does as you would expect, it cancels anything the environment is currently doing in the background and closes all the windows. The ‘Quit’ function does not try to save any of the programmer’s work; since this is a live programming environment, the environment constantly saves the programmer’s work to a file.

The slider elements are what allow the programmer to move forward and backward in their program’s execution. The left slider acts as the start point for what information is shown, it acts as a minimum value that says ‘do not show me any information before this point.’ The idea for this is that as a project grows, the information in a program can become overwhelming and being able to hone in on a specific area by setting the start and end points can bring clarity to a large program. The right slider does the opposite of the left slider because it enforces the end point for what information is shown, acting as the maximum value. This is where the programmer has complete control over time to limit the information that they see.

The last element in the Menu field is the ‘Input File’ button, which is tied to the Input box. The Input File button opens up the native file browser on the programmer’s operating system for the programmer to select an input file to their program. The contents of the input file are then inserted into the Input box.

3.2.2 Source Code Box

The Source Code box is where the programmer will spend most of their time and it provides the most information about the programmer’s code. The Source Code box is first and foremost where the programmer types their code that will be run by

the environment. As standard in many text editors and IDEs, the source code box performs syntax highlighting on the programmer's code. Several features have been added to the Source Code box that provide additional information to the programmer about their code and that allow the programmer to perform additional actions. The features, discussed in greater detail below, include:

- Line Highlights: show the result value for each line
- Clickable Loops: allow the programmer to walk through a loop
- Clickable Functions: allow the programmer to view previous calls and perform unit tests
- Clickable Classes/Objects: allow the programmer to perform unit tests
- Error Handling: show where errors occur

3.2.2.1 Line Highlights

Line Highlights are an essential feature that quickly provide the results of a line of code to the programmer. As seen in Figure 3.2, the highlighted line

```
b = 1 + 1
```

has the resulting value of 2. Although this is a relatively simple line, this feature is quite helpful as the line's complexity grows with multiple variables whose values are not constant.

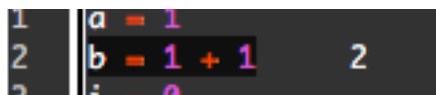


Figure 3.2: Basic Line Highlight

Another feature of line highlights is for programmer defined functions. When the programmer declares a function and then calls that function, the calling line (when

highlighted) will show the results for its own line and for the lines of the called function. The function lines will be highlighted and the parameters from the calling line will display the proper results for the function lines. This can be seen in Figure 3.3, where line 12 is highlighted by the programmer; the arguments to the function call `add` are `a=1` and `b=2` and the result of the call to `add(a, b)` is 3. If this result was unexpected, the programmer can examine the result of each line in the function to see where it went wrong. This feature also works across multiple programmer defined

```

7
8  def add(x, y):      y=2, x=1
9      z = x + y      3
10     return z        3
11
12  c = add(a, b)      3
13  d = add(i, i)

```

Figure 3.3: Advanced Line Highlight

functions as seen in Figure 3.4, where highlighting line 9 results in all called functions and their resulting values being shown.

Line Highlights provide the most information to the programmer about their code in an easy and accessible format. Line Highlights include the result for each line and the lines of any called functions.

3.2.2.2 Loops

The environment allows the programmer to walk through a loop by clicking on the line of code that starts the loop. Once the line is selected, the environment creates a pop-up dialog box, similar to the ones seen in Figure 3.5.

The pop-up box has three main parts: a slider value, the loop's code, and the results for that code based on the current iteration. The slider value allows the

```
1  def add(x,y):      y=2, x=1
2      z = x + y      3
3      return z        3
4
5  def mult(x,y):      y=3, x=3
6      z = x * y      9
7      return z        9
8
9  c = mult(3, add(1,2))  9
10
```

Figure 3.4: Advanced Line Highlight with Multiple Function Calls

programmer to move forward and backward through the loop. The loop's code is a direct copy of the code in the source code box. The resulting values change dependent on the current iteration.

In Figure 3.5, there is a simple loop which prints the value of i five times. In Figure 3.5a, the current iteration is at 0, which is always the first iteration shown when the pop-up box is shown. As the programmer walks through the loop, as seen in Figure 3.5b, the resulting values are updated based on the current iteration. Lastly in Figure 3.5c, the terminating iteration of the loop is reached.

This feature can be extremely useful for beginners as loops can be a rather difficult concept to master and being able to easily see what happens at each iteration in a controlled manner is a vital part of making information accessible to the programmer.

3.2.2.3 Functions

Functions are the backbone of programs and mastering them can be difficult for beginners. To help programmer's have a better understanding of the functions in their code, the environment allows the programmer to click on the function definition. The environment then creates a pop-up dialog box for the programmer to interact with,

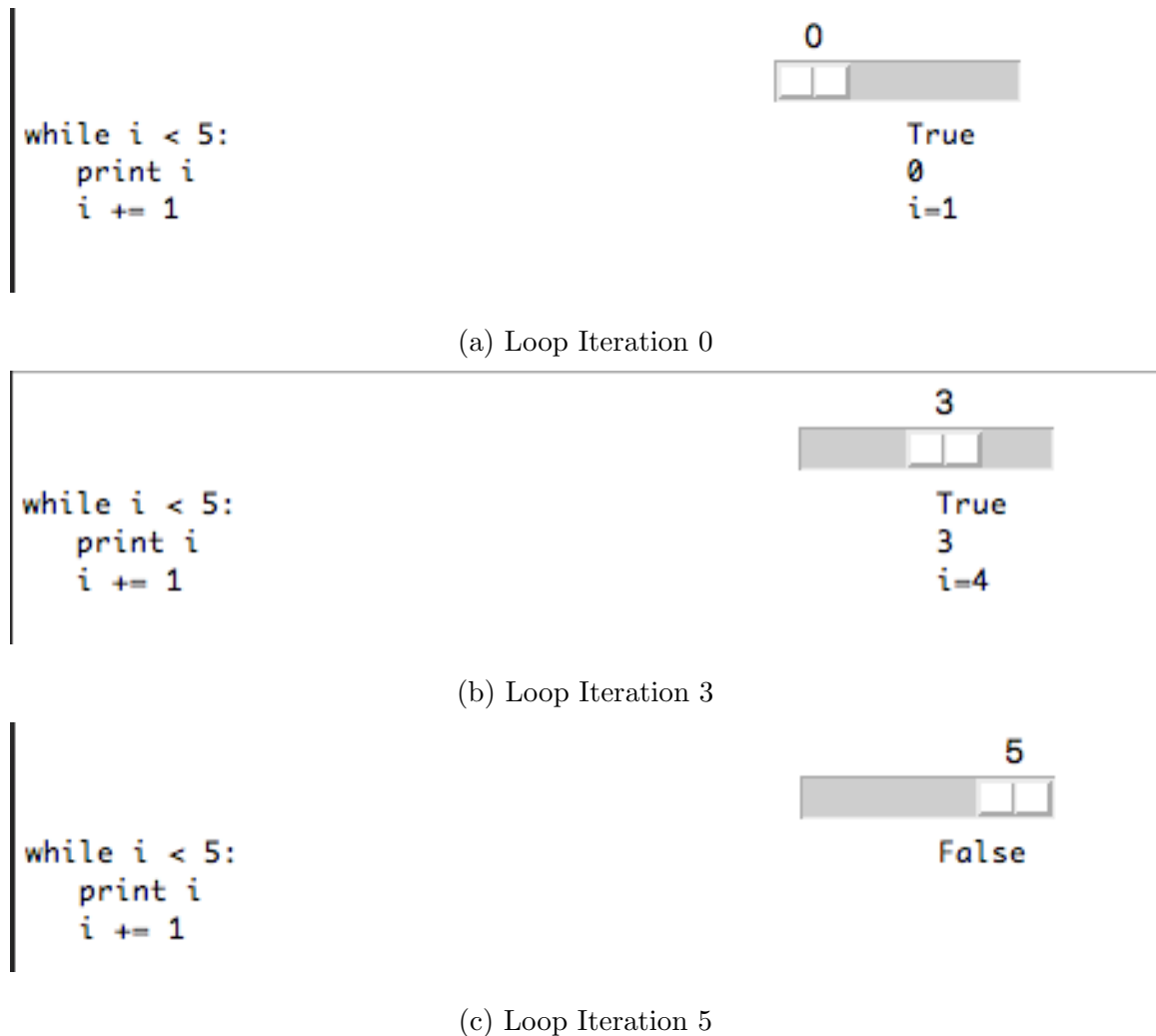


Figure 3.5: Clickable Loop Iterations

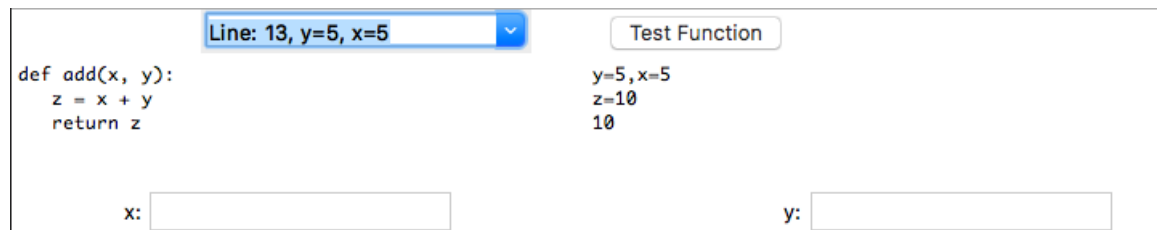
similar to the ones seen in Figure 3.6.

The pop-up dialog box serves two purposes. First, it allows the programmer to view calls that have been made to that function in their code. Second, the pop-up box allows the programmer to quickly unit test a function by entering values for the parameters to a function. The pop-up box has five main parts: a drop down menu of function calls, a 'Test Function' button, the source code of the function, the resulting values for the function based on the parameters, and a parameter section.

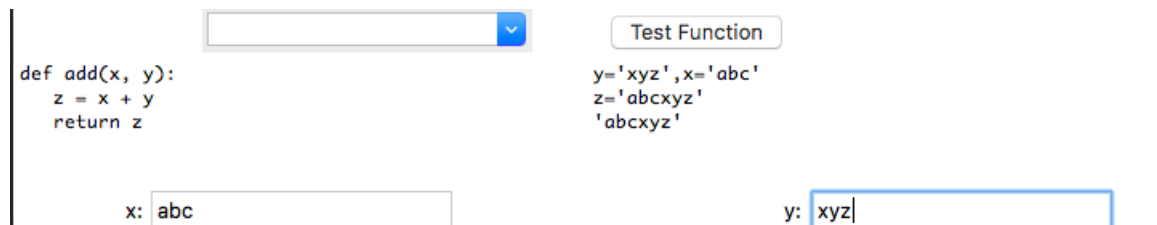
The drop-down menu has an entry for each call that is made to the function in

the source code. Each entry contains the line number of the call and the arguments passed to the function. If the function call occurs in a loop, there will be an entry for each call made in the loop with an indicator specifying the call number at that line.

The ‘Test Function’ button tells the environment to run the function’s code with the specified parameters. In the parameter section an input field is displayed for each parameter in the function. An example of viewing a called function can be seen in Figure 3.6a, where the calling line is line 13 and the parameter values are x=5 and y=5. This is an important feature that allows the programmer to jump between all the function calls for this function to ensure the function performs as expected. The other feature in the pop-up box, as seen in Figure 3.6b, allows the programmer to unit test a function. This is important because the programmer can quickly test their function for the standard and edge cases of a program and even try things that may not have been originally intended for the function to handle.



(a) Viewing Function Calls



(b) Unit Testing a Function

Figure 3.6: Clickable Functions

Before moving on to the next section, it is important to note that Class/Object functions support a simplified version of the pop-up dialog box. Clickable Class/Ob-

ject functions only support the first feature mentioned above, which is viewing function calls that are made in the source code and the results of those calls. The pop-up box does not support unit testing on Class/Object functions because Class/Object functions require that an object is created and initialized before any functions can be called on that object. Unit tests on a function inside of a Class/Object, are described in the next section.

3.2.2.4 Objects

Objects provide a programmer with a lot of flexibility to create new data structures. The environment provides a pop-up dialog box to unit test the class and its functions. The pop-up box has four main parts: a ‘Test Class’ button to run the unit test, the source code for the class, the results from the unit test, and a function/parameter section.

The source code for the class is taken from the programmer’s source code. The function/parameter section is similar to the parameter section for Functions, but this section has an entry for any parameters to the `__init__` function for the class. The parameters must be filled before any unit test can be run. As seen in Figure 3.7, the class has one parameter for the `__init__` function that is filled and the results of the test are shown on the right. The results, unlike before, do not line up with the source code. Instead, the results first show the function that is called then the results for each line in that function associated with its line number. The reason for the change in how results are shown to the programmer is due to the added feature that allows the programmer to unit test a class object and its functions as described below.

One important part about classes, is that the programmer can add functions to a class and those functions need to be tested on an object. In the parameter/function section there is a drop-down menu that has all the callable functions in a class. Once

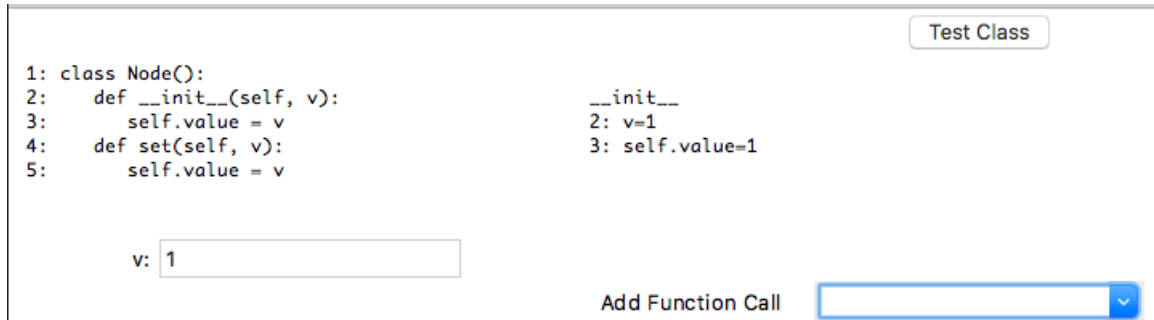


Figure 3.7: Basic Class Test

a function is selected, a denoting function name label will be appended to the bottom of the pop-up box and entries will be created for the parameters to the selected function. The programmer can then enter those parameters and test the function as seen in Figures 3.8a and 3.8b. These figures show a function that has been selected twice from the drop-down menu. This results in two more sections with (as described above) a function label and the parameters for that function, once per call to the function. The results of running the unit test are seen in Figures 3.8a and 3.8b. The ‘`__init__`’ function is called first to create the object; the evaluation of each line of ‘`__init__`’ is shown in Figure 3.8a. Figure 3.8b shows the evaluation of two separate calls of the ‘`set`’ function. As more and more functions are added to the unit test, their function labels and parameter lists are appended to the bottom of the pop-up box, which provides a lot of flexibility for the programmer to test any mixture of function calls on their object.

3.2.2.5 Error Handling

Beginning programmers are prone to make mistakes and create errors in their code that violate Python’s rules. It can also be very hard for beginner’s to track down these errors, which is why it needs to be clear where errors occur in the programmer’s code. The environment handles two types of errors: compiler errors and runtime

Test Class

```

1: class Node():
2:     def __init__(self, v):
3:         self.value = v
4:     def set(self, v):
5:         self.value = v

```

```

__init__
2: v=1
3: self.value=1

set
4: v=5

```

v:

set:

v:

set:

v:

Add Function Call

set

▼

(a) Advanced Class Test

Test Class

```

1: class Node():
2:     def __init__(self, v):
3:         self.value = v
4:     def set(self, v):
5:         self.value = v

```

```

set
4: v=5
5: self.value=5

set
4: v=7
5: self.value=7

```

v:

set:

v:

set:

v:

Add Function Call

set

▼

(b) Advanced Class Test (scrolled)

Figure 3.8: Clickable Classes/Objects

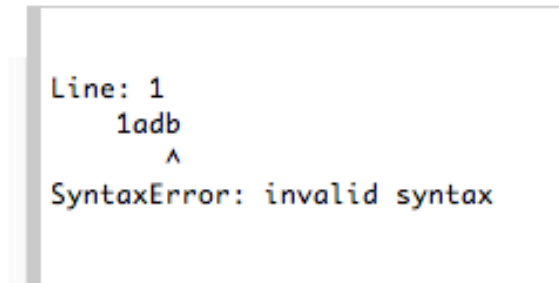
errors.

Compiler errors occur before any code has been run, when the programmer's code violates the syntax rules of Python. As seen in Figure 3.9a, the offending line in the Source Code box is highlighted red to indicate an error on that line. In Figure 3.9b, a message is also displayed in the variables box that provides the line number and the error message from Python's compiler.

Runtime errors occur while the code is running and are the result of bugs in



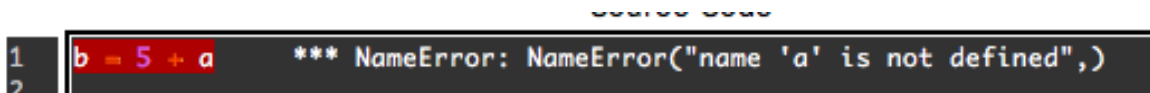
(a) Source Code Error Highlight



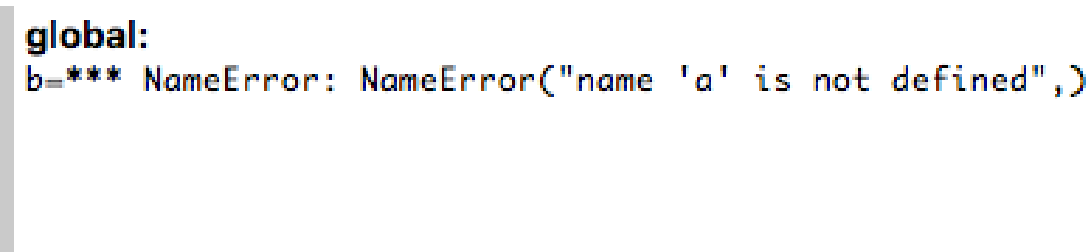
(b) Variables Box Notification

Figure 3.9: Compiler Error

a program. As seen in Figure 3.10a, the offending line in the Source Code box is highlighted red to indicate an error has occurred on this line during execution. When highlighting over this line, there is some additional information provided about exactly what went wrong on the line. To make it even more clear, the environment uses the Variables box, as seen in Figure 3.10b, to specifically show which variable caused the error, however this only applies when a variable is involved. Otherwise, the only indication of an error is in the Source Code box with the red highlighted line.



(a) Source Code Error Highlight



(b) Variables Box Notification

Figure 3.10: Compiler Error

3.2.3 Variables Box

The Variables box keeps track of all variables that are declared in the programmer's source code and the scope in which they are declared. This feature is somewhat similar to the stack frame and stack frames found in Python Tutor and PyCharm. The way the Variables box works is that it keeps the latest values for each variable and maintains all scopes, such as previous function calls, as seen in Figure 3.11. If the programmer changes the menu sliders forward or backward then the Variables box changes the values of the variables. The Variables box is a place where, at a glance, the programmer can quickly glean the state of all the variables in their program.

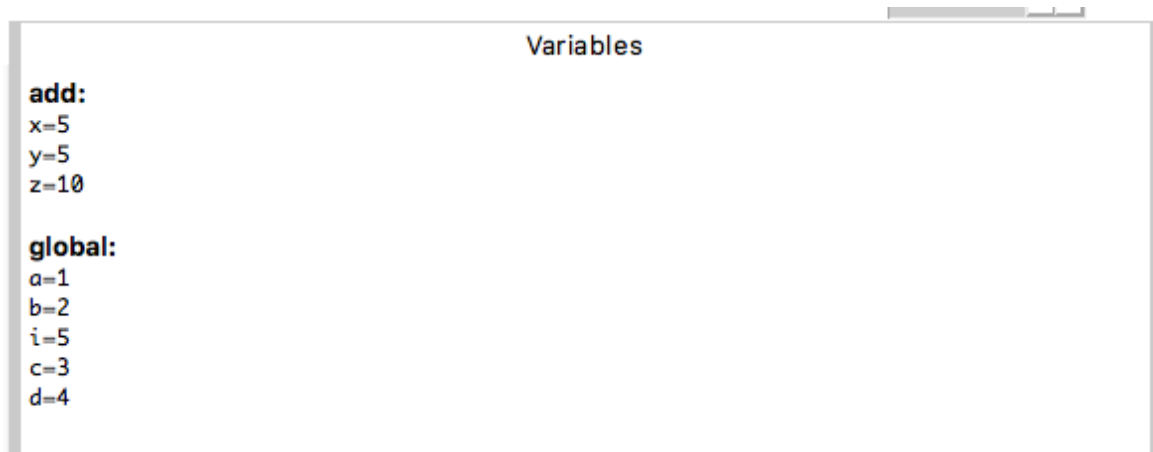


Figure 3.11: Variables Box

3.2.4 Objects Box

The Objects box creates a visualization for data structures that are created via programmer defined classes. Many introductory courses require programmer's to create list, tree, or mapping structures by defining the classes and filling them with objects. These data structures are a vital tool in programming, but it is difficult for a programmer to keep a visual representation of the structure in their head. These data structures can also become quite complex and hard to manage, which is why it

is important to create a programmer friendly representation of their structure that also provides information about the structures. The Objects box attempts to draw such a graph for the programmer like that shown in Figure 3.12, which is a simple binary tree structure.

The Objects box has two main parts: a drop-down menu of all programmer defined objects and the area meant to hold the visualization of the data structure. The drop-down menu contains variables from the programmer's source code that reference a class/object. Once a variable is selected, a visualization is shown for that variable and any objects it points to. Since some objects can be a part of a data structure and never have a variable attached to them, there should still be an easy way for the programmer to refer to these objects. Behind the scenes these objects are created with an instance id, however these ids change based on when the program is run due to available memory and are not simple to measure. Therefore, objects are provided with a simple id by the environment, based on the order in which they are created. Objects without any variables referencing them are given names based on their class name and simple id. As seen in Figure 3.12, there is a variable head that is the root of the tree. The object that is referenced by head also has the id of 0 as seen in Figure 3.13. The remaining nodes however, use the environment's naming scheme. As seen in Figure 3.12, there are arrows and descriptive names next to those arrows that point to objects. The names next to the arrows are the variables inside the object, showing that in object head, the left variable points to Node_1.

To understand what is happening inside each object, the environment allows the programmer to click on these different objects in the graph. Figure 3.13 shows information about the 'head' object. The environment provides the Class name of the object. The environment provides any names that the object has in the source code, these are variables that point to this object. The id field is the same as discussed above, it is a unique simple id for the object based on the order of its creation. The

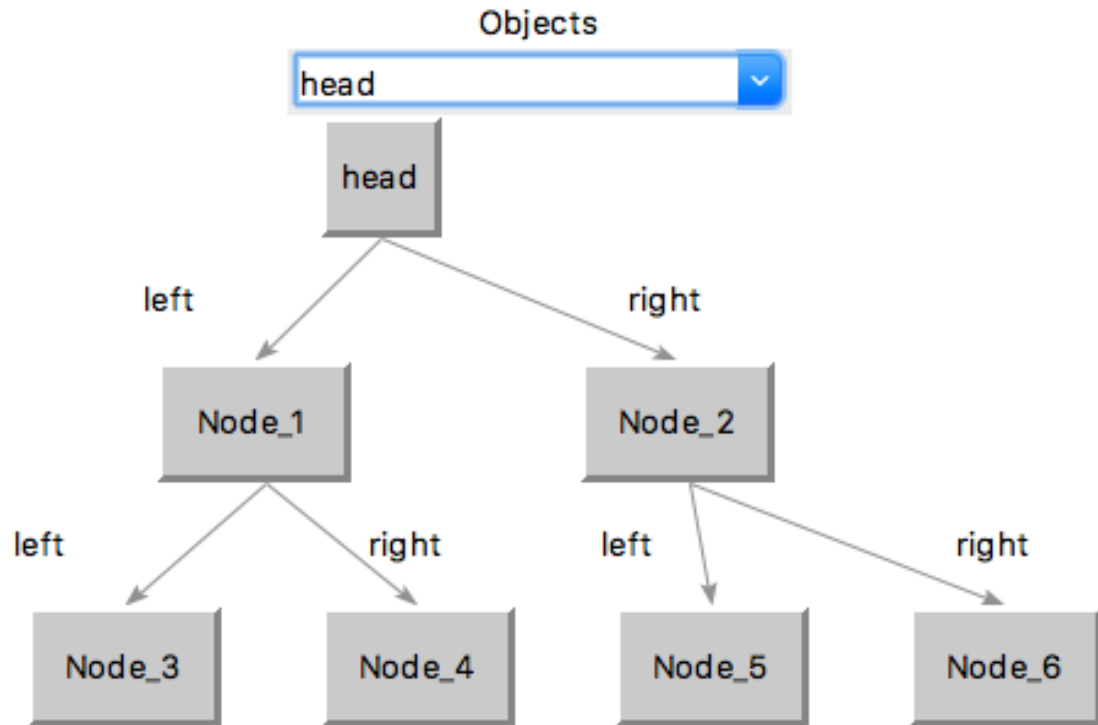


Figure 3.12: Objects Graph

Variables field provides information about all object variables and their values for the object. For objects with functions, the Functions field shows information about object functions that have been called and any variables in those functions with their values.

Creating a graphical representation of programmer defined data structures helps the programmer understand what their data structure looks like. The representation also allows the programmer to get a closer look into their individual objects to ensure they are correct.

3.2.5 Input Box

The Input box handles all input that goes into a program, similar to the command line. Each input is delimited by a newline, which represents the programmer hitting

```
Class Name: Node
Name: ['head']
ID: 0
Variables:
    right=Node_2
    value=0
    left=Node_1
Functions:
    set:
        v=0
    set_left:
        l=Node_1
    set_right:
        r=Node_2
```

Figure 3.13: Object Information

‘enter’ on the command line. The programmer can modify the input at any time and those new changes can be seen in the code’s output immediately. The programmer does not have to rerun the program for all the different inputs they want to test, but they can quickly see if a change to one input will still work for their program.

The environment saves the programmer’s input, so that each time they open the environment they do not have to type in the input. A key aspect to the input box is that it does not take away from the live programming experience. Normally the user would have to enter the different inputs in the terminal for each run of the program or attach some input file, but that work is done for the programmer by the environment. Once they have entered their input, the environment will use those inputs to keep running the program as the programmer continues to write it. As mentioned above, the programmer can also select an input file, which will overwrite the current contents of the input box with the new inputs.

One feature was added in response to a survey participant (discussed in Chapter

5). The participant noted that when invalid input is entered in the input box, the environment silently fails and does not provide any information to the programmer. To fix this issue, the environment now does three things to alert the programmer to an issue with their input, as seen in Figure 3.14. Figure 3.14a shows that the input in the input box is highlighted red to signify an error with that input. Figure 3.14b highlights the line of code that accepts the input from the input box. Figure 3.14c shows an error message about the input in the variables box.

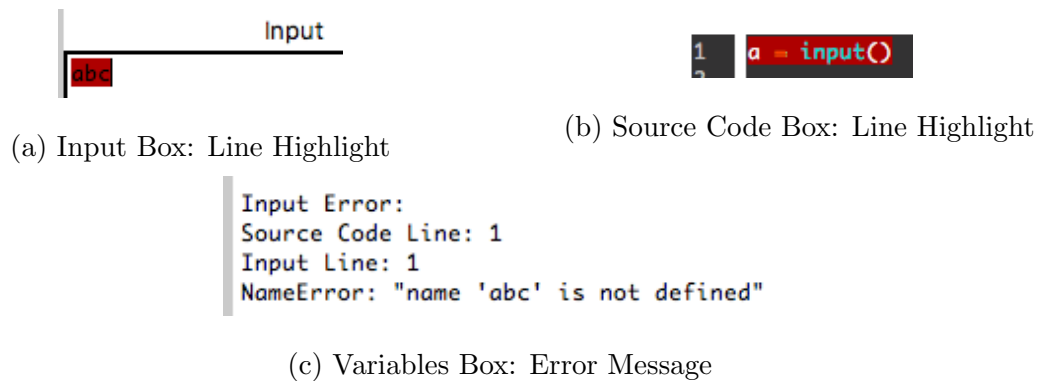


Figure 3.14: Input Correction

3.2.6 Output Box

The Output box is fairly straightforward, but a vital part of any project is when a program requires some output to the terminal. This is where the output box comes into play, any print statements or other output from the program are directed to the Output box.

Chapter 4

IMPLEMENTATION

The backend of the environment uses Python’s Abstract Syntax Tree [15] and the built-in Python Debugger [13] to generate the information needed for the GUI side of the environment. When the backend is called on by the GUI, it performs a three step process: walking the abstract syntax tree, running the programmer’s code, and displaying the results. Each of these components is covered in the sections below.

4.1 Walking the Abstract Syntax Tree

The first step in presenting information to the programmer is to understand their code and determine what information can be made available to them. To do this, it is important to analyze each line of code and determine how the computer views that code and how the computer breaks down each line of code into smaller parts. The way this is typically done is through an Abstract Syntax Tree (AST). The AST is a compiler’s/interpreter’s internal representation of a program. Python has a built-in AST module [15] that allows for easy manipulation of the tree. The AST module lets the user visit each node in the tree and access each part of Python’s Grammar. The Grammar defines the syntax of a programming language and dictates the nodes of the AST. The AST module [15] allows the environment to generate an AST of the programmer’s code and then traverse that tree. This is an important step in gathering information about the programmer’s source code, because while traversing the tree, the environment can use the rules of Python’s grammar to store information about the source code.

The way this works at a high level is that the environment takes the programmer’s

source code and uses Python’s ast module to create the AST from the source code. The environment inherits from Python’s ast.NodeVisitor class to walk the tree and gather information about each line of code. The main pieces of information gained are about individual lines of code, function/loop/class lines, and variable scope. The information gained from individual lines of code is about the type of statement that line is and the expressions used in that line of code. An example of a statement is `b=a+1`, where the expressions for that statement are `[a, 1, and (a+1)]`. Variable scope helps the environment keep track of where variables are declared in the code.

Consider a very simple example with two lines of code.

```
1 a = 1
2 b = a + 1
```

The first step to analyze a program involves using Python’s ast module to build the AST from the two lines of code. The AST for those two lines is rather complex, as seen in Figure 4.1. The ‘Module’ section has one element, which is ‘body’, that is a list of every statement. There are many types of statements, but for the simple example there are two ‘Assign’ statements. The ‘Assign’ statement has an element for ‘targets’, which is a list of names that need to be assigned a value. The ‘value’ element in the ‘Assign’ statement is an expression. Expressions can take on a wide variety of values, such as Variables, Numbers, Strings, Operators, and so on. The first ‘Assign’ statement represents the first line of code (`a=1`); its ‘targets’ field has one Name expression for the variable ‘a’. The first ‘Assign’ statement also has a ‘value’ field, which is a Num expression for the number ‘1’. These two expressions represent the ‘a’ and ‘1’ from the statement. There is nothing that denotes the ‘=’, because it is inherently assumed to be part of an ‘Assign’ statement. The second ‘Assign’ statement (`b=a+1`) again has one Name expression in its ‘targets’ element that identifies the variable ‘b’. The ‘value’ field in this second statement is more complex, because it represents a ‘BinOp’ (a binary operator). A ‘BinOp’ has a left


```

Module(
  body=[
    Assign(
      targets=[
        Name(id='a', ctx=Store())
      ],
      value=Num(n=1)
    ),
    Assign(
      targets=[
        Name(id='b', ctx=Store())
      ],
      value=BinOp(
        left=Name(id='a', ctx=Load()),
        op=Add(),
        right=Num(n=1))
      )
  ]
)

```

Figure 4.1: Simple AST Example

side expression, an operator, and a right side expression. In the case of our second statement, the left side is for the Name expression that is the variable ‘a’, the operator is ‘Add()’ or the ‘+’ symbol, and the right side is for the Num expression that is the number ‘1’.

For the two lines above, the environment is able to gather information by walking the AST. For the first line (a=1), the environment gathers:

- Assigned Variable: **a**
- Expressions: **1**
- Statement Type: **assignment**

For the second line:

- Assigned Variable: **b**

- Expressions: **a**, **(a+1)**
- Statement Type: **assignment**

As mentioned, the environment also gathers the variables scope, in this case, the global variables are [a, b]. This is all the information needed from the AST for the two lines of code above.

The environment needs to be able to tell the programmer what happens at each line of code. By visiting each element in the tree, the environment keeps track of expressions at each line of code that will be evaluated later to determine their value. This is why the first step in getting information about the programmer's source code is focused on finding all the expressions for each line of code. After all the expressions are found the next step, detailed in Section 4.2, focuses on evaluating all the expressions found in this step and storing the results.

4.2 Running the Programmer's Code

To run the programmer's code, the environment makes use of Python's built-in debugger (PDB). A debugger is a tool that helps the programmer view a line of code and its execution in the context of the program as a whole. To display information to the programmer about their code as it executes, that code must be executed in a controlled manner, which is where PDB comes in [13]. PDB allows the environment to walk through the programmer's code and execute each line of code while making evaluations about the information gathered from the AST, so that the results can be shown to the programmer. This step has two parts: getting the evaluated expressions and the value of each variable for each line. The first part is so that the environment can display information about each line to the programmer. The second is so that as the programmer walks through their program, the variables display their proper values at that point in execution in the Variables box. The next example includes the

same two lines of code from the previous section and builds on that section's results.

```
1 a = 1
2 b = a + 1
```

The environment evaluates the expressions gathered from walking the AST (like the ones in the previous example), then stores that information to make it available to the third step (Displaying the Results). For the first line of code in the example, the results of stepping to that line and evaluating the expressions are:

- Assigned: **True**
- Values: **1: 1**
- Result: **a=1**

The first item, 'Assigned', tracks that an assignment occurred in that line of code. The 'Values' item holds all the evaluated expressions' results. The 'Result' value is the most important as it is the final value for that line and will be used to tell the programmer what that line of code did. The second line has two expressions and therefore two values. The 'Result' item is simplified to show just the final value for the line with its assignment.

- Assigned: **True**
- Values: **a: 1, (a+1): 2**
- Result: **b=2**

The environment runs the code by creating a second process to execute PDB with the programmer's code. The environment communicates with PDB to step through the programmer's code and for each line uses the information gathered from walking the AST to evaluate expressions in the right context. To do this the environment sends a command to the second process with the expression it wants evaluated and

then stores the results from the second process. To accomplish this interprocess communication the environment uses a pair of pipes to pass messages. PDB was slightly modified to allow for the use of pipes instead of sending its output to the console. The environment is able to loop through all the expressions for a line of code and have PDB do the hard work of evaluating them. However, there are still some special cases that the environment has to handle: program input and expressions that modify the state of a program.

Program input is used in many early programming exercises to test a program and show that a program can perform a variety of tasks. However, when a program is executing and comes to a point where it needs input, it will halt and wait for that input. Therefore, the environment needs to be able to get input from the programmer and pass it to the waiting process. To do this, the process that controls the debugger has to set an event flag waiting on the programmer to enter their input. Once the environment has that input, it stores it in a shared variable and changes the status of the event flag, so that the waiting process can then use that input with PDB and continue executing the program.

The second special case, expressions that modify the state of a program, is a bit more intriguing. In most cases evaluating an expression does not cause any changes to occur in the running program, because they are merely asking for the value of an expression. Attributes, a specialized expression, on the other hand are quite different, especially when evaluated inside PDB. Different attributes that are evaluated on an object occur as if the attribute was a line of code in the program.

Consider the List type in Python. A List object has many different attributes, better known as methods, that can be called on that object, such as the `append()` attribute. `Append()` allows the programmer to add an item to the end of the list. Say that a line of code in the programmer's program was: `list.append(1)`. After this

line of code occurs, there should be a 1 added to the end of the list. When the environment runs this line of code a 1 will be added to the list and the environment will get three expressions to evaluate: list, 1, and list.append(1). The environment will evaluate these three expressions and the first two cause no problems. However, the third expression, a method invocation, when evaluated actually executes on the object in memory while the program is running, therefore a second 1 is added to the end of the list. This now poses a major problem for the environment, it has completely altered the state of the program and skewed the programmer's results.

To fix the altered program, there are two solutions: open a third process or write a Python interpreter to handle these special cases. The environment opens a third process. The job of the third process is to force the second process to maintain the program's integrity and not allow the environment to modify the program's execution. The third process is again running PDB with the user's code, however this process has a slightly different job. While the second and third processes both step through the program at the same time, the third process is responsible for only evaluating the value for each variable at each line. Since walking through the AST, the environment knows all of the variables and their scope. The third process is responsible for asking PDB the value of each variable. Once the third process has gotten all the variable values, it uses those values and executes statements on the second process. These statements are assignment statements for variables and their correct values, because the third process does not perform any modification to the state of a program. From the example: list.append(1), in the second process our list has a duplicate 1 added to the end, however in the third process the variable 'list' is correct. Therefore, by executing the statement: list=[1] in the second process, the environment fixes the error that has occurred and is still able to evaluate the various expressions. See Figure 4.2 for a visualization of what is done to solve this issue. As can be seen, the red highlighted text shows where the error occurs and the arrow shows how the third

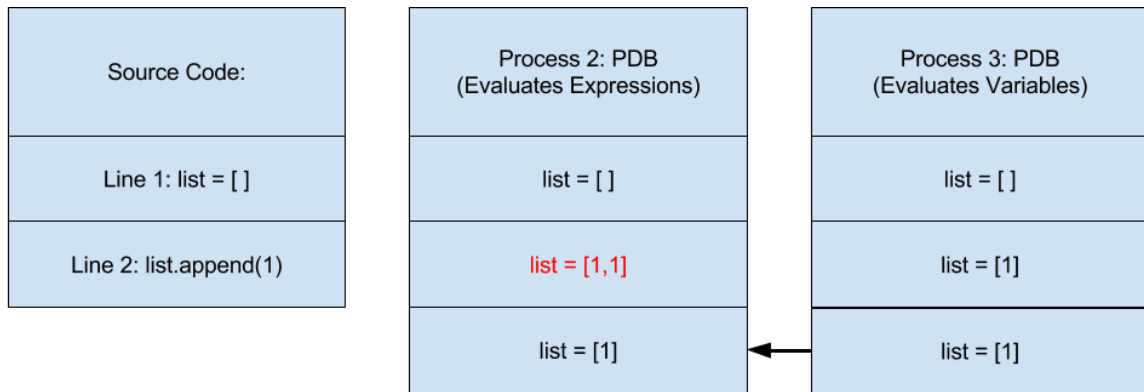


Figure 4.2: Variable Correction

process corrects the error in the second process. This error can occur for any object: lists, dictionaries, programmer defined objects, etc.

Once the environment has finished evaluating all the expressions and variables in the programmer's code it can use this information to display the results to the programmer.

4.3 Displaying the Results

Getting the results to the user requires the environment to make the information accessible to the programmer. As discussed above about how the GUI operates, this section discusses what happens behind the scenes to make this happen. The environment runs through a looping function to maintain the state of a live programming environment every few hundredths of a second.

The first step taken by the environment is to highlight the programmer's source code. To do this the environment uses a third party Python package called Pygments [11]. The environment uses Pygments' lexer for Python. The lexer is used to split the source code into various tokens. These tokens are specific pieces of code, such as keywords, variables, etc. Certain tokens are then used to tag the source code in the

text box. For any text that has a specific tag, its color is then changed.

To display the results of each line when the programmer highlights a line, the environment traverses the results gathered in the previous section. The environment then tags each line with the information retrieved from running the code. These same tags are used to create clickable loops, functions, and classes.

The objects box requires a bit more work to be created, the environment creates a generic object class to hold information about the objects created by the programmer. These generic objects hold the class name, variable name, id, class variables, class functions, and class function variables. The visualization of the object's structure is based on a tutorial in Mark Lutz's book: *Programming Python*, 2nd Edition [8]. The tutorial shows how to create a generic tree object viewer [8]. The tutorial has examples for binary trees, which were modified to match the generic object structure that the environment used. The object viewer was also modified to enhance the visualization of the structure.

All of these steps take place behind the scenes to make the information gathered from walking the AST and running the code accessible to the programmer. The GUI is able to take the results and show them to the programmer.

Chapter 5

VALIDATION

To measure the utility of the live programming environment I developed a user study to allow beginners to assess the environment to determine if it is beneficial. Two studies were performed: one was proctored, one was online with participants working without support.

5.1 Survey Overview

```
1  a = 1
2  b = 1 + 1
3  i = 0
4  while i < 5:
5      print i
6      i += 1
7
8  def add(x, y):
9      z = x + y
10     return z
11
12 c = add(a, b)
13 d = add(i, i)
14 print c
15 print d
16
17 a = input()
18 b = input()
19 c = input()
20 d = input()
21
22 print a
23 print b
24 print c
25 print d
```

Figure 5.1: Survey Code

The goal of the survey was to determine if the environment and its features are helpful for beginners learning to program. The survey was performed with consent

from the Institutional Review Board at Cal Poly [5]. A summary of the survey results is provided below and the complete survey can be found in the Appendix. The participant writes a simple program, as seen in Figure 5.1, and is asked to interact with different features in the environment. The participant is provided the source code in the survey that they must then type into the environment and then they are asked to interact with a specific feature, such as Line Highlights, before moving on to the next part where they may enter more code or interact with a different feature. After the participant completes the tasks in the survey they are asked to fill out a questionnaire about their experience. The questionnaire consists of multiple choice questions answered on a scale from 1 to 5 (extremely unhelpful to extremely helpful), each with a comment section available. The questionnaire contains seven sections that each focus on a particular feature in the environment. Following is a breakdown of each section in the survey with a discussion of the questions asked of the participants, their responses, and any comments left by the participants. The seven sections address the overall environment, source code box, variables box, changing program state, input box, output box, and a final comment section. The entire survey was designed to take less than one hour to complete.

5.1.1 Student Survey

Students in introductory courses for programming at Cal Poly were asked to participate in the survey. Many introductory courses at Cal Poly recommend that students use PyCharm as their IDE. Although PyCharm is very sophisticated in the features it offers, students may benefit from the many features implemented in a live programming environment. The features the students were asked to interact with cover all the features discussed in the implementation chapter except for classes, because students in introductory courses had not yet learned about classes at the time of the survey. In the first administration of the study, I was present during the

period when students completed the survey. For the second study, students were able to complete the survey on their own time either on campus or at home.

5.2 Proctored Survey Results

The proctored survey was completed by 13 participants in Cal Poly's Fundamentals of Computer Science I course. The survey was administered during the 5th week of Spring 2017. Before the survey was administered, participants were given a demonstration of the environment that introduced its different features.

5.2.1 Overall

The participants were asked general questions about their experience with the environment. Below is the list of questions with the responses shown in Figure 5.2.

Q1: Overall did you find the Environment helpful?

- Average Score: 4.77

Q2: Would you use the Environment when coding?

- Average Score: 4.23

Q3: Are the Environment's aesthetics straightforward? (easy to understand)

- Average Score: 4.54

Q4: Was it easy to use the Environment?

- Average Score: 4.38

Participants were pleased with the environment (Q1: 4.77) and would use it when coding (Q2: 4.23). Students found it relatively easy to understand (Q3: 4.54) and use (Q4: 4.38). Some of the comments received from the participants can be found below:

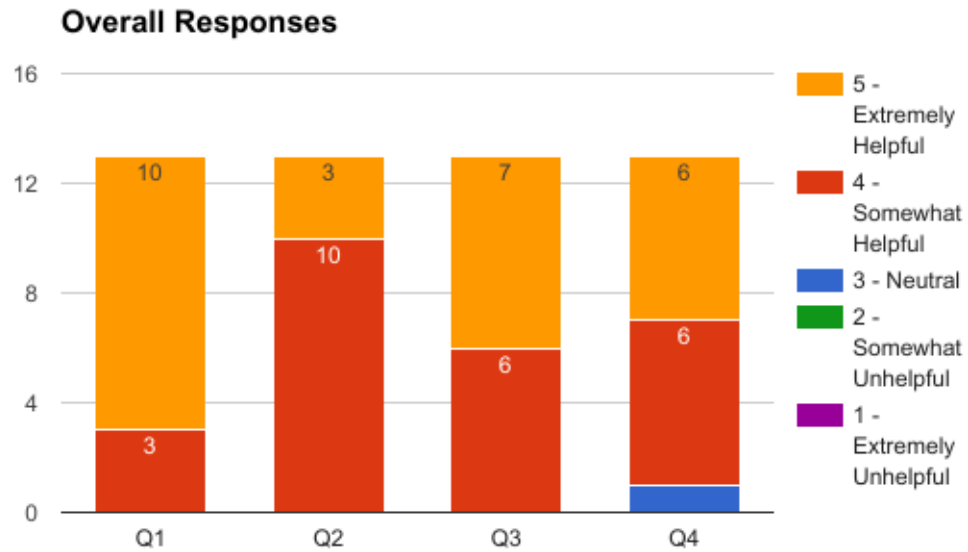


Figure 5.2: Proctored: Overall Responses

- “When can I download a copy for my computer? I need help in this class lol.”
- “Very helpful for someone like me, who usually has to google every line of code I’ve written to see if it works/what it does”
- “Its much much easier than clicking back and forth between things like pyCharm and the terminal.”
- “I am impressed by this! Make it look a bit more professional and I can see this being an actual product especially for educational purposes.”

The overall feedback from the participants shows that they did indeed find the environment helpful, but would prefer if the environment had a more appealing GUI.

5.2.2 Source Code Box

The source code box has the most features in the environment and the participants were asked questions about their experience with each of the following features: line

highlights, clickable loops/functions, and unit testing a function. Below is the list of questions and average scores with the responses to these questions shown in Figure 5.3.

Q1: Did you find hovering over a line of code and the additional information shown helpful?

- Average Score: 4.62

Q2: For lines of code that called a function (such as ‘add’). Did you find it helpful that the function lines were also highlighted?

- Average Score: 4.77

Q3: Clicking on a Loop: Did you find it helpful to be able to click on a Loop and walk through it?

- Average Score: 4.54

Q4: Clicking on a Function: Did you find it helpful to be able to click on a Function and examine previous calls to that function?

- Average Score: 4.77

Q5: • Average Score: 4.85

The participants were very pleased with the features added to the source code box, as seen in Figure 5.3, with high average responses for each: Q1 - 4.62, Q2 - 4.77, Q3 - 4.54, Q4 - 4.77, and Q5 - 4.85. Some participant responses for the various features can be found below. One comment expressed an issue with syntax highlighting in the source code box: “the last letter in keywords are not colored.” However, this issue could not be recreated and its cause for the one participant is unknown.

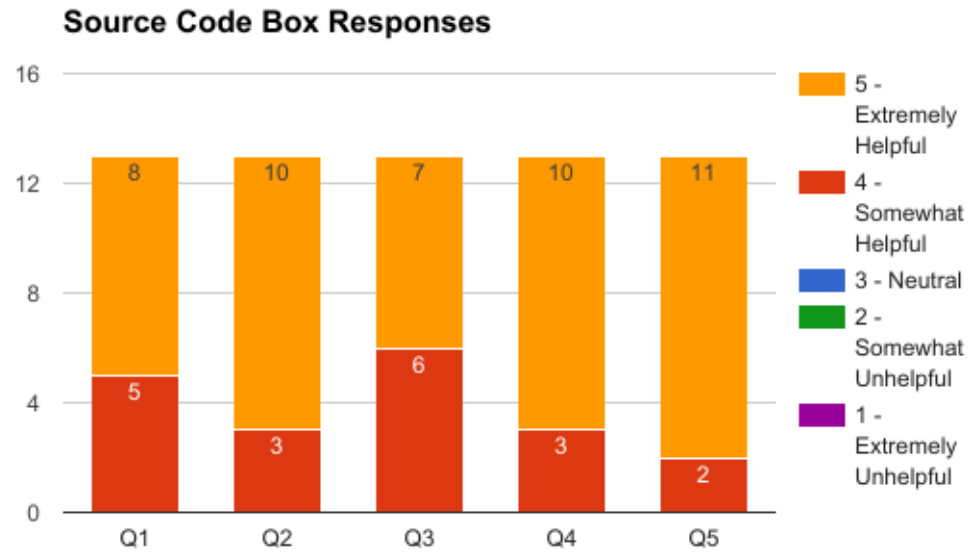


Figure 5.3: Proctored: Source Code Box Responses

5.2.2.1 Clickable Loop Comments

- “easy to notice when loop is not working as intended”
- “nice to see how far I can take the loop”

5.2.2.2 Clickable Function Comments

- “nice for debugging”
- “SO much nicer than having to write a test case for every function”

5.2.3 Variables Box

Participants were asked about their experience with the variables box. Below is the list of questions and average scores with the responses to these questions shown in Figure 5.4.

Q1: Did you find it helpful to see the state of variables?

- Average Score: 4.33

Q2: Did you find it helpful to have variables separated by their scope? (Global/-
Functions)

- Average Score: 4.38

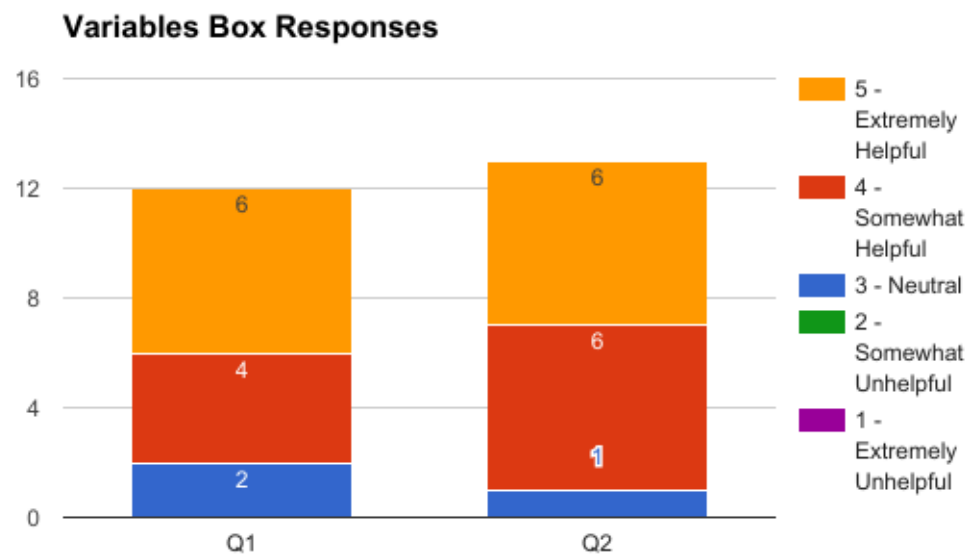


Figure 5.4: Proctored: Variables Box Responses

The variables box has more varied responses, but it is still generally viewed as positive and considered useful by the participants, as seen in the average scores: Q1 - 4.33 and Q2 - 4.38. From the comments below, the Variables Box was well received by some of the participants.

- “good to know which vars are global”
- “less lit [exciting] than other features but still lit [exciting]”
- “I liked this feature the most”

5.2.4 Changing Program State

Changing program state is when the programmer is able to move forward/backward through their code. The participants were asked to move the slider to see how it would affect the information shown by the environment. Below is the question and average score with its responses shown in Figure 5.5.

Q1: Did you find it useful to move forward/backward through time via the Sliders and change how much information is shown?

- Average Score: 3.77

The participants found the sliders to have a small impact on their experience, with an average score of 3.77, due to the high number of neutral responses. This could be because the program in the experiment is rather small or because the source code box shows the result for each line acting as a similar feature to the one created by the sliders. One comment suggests that some might find it difficult to use the Slider's feature, as seen in the comment below.

- “Sort of confused on how to use this properly, so I put neutral but I can see how it can be extremely helpful.”

5.2.5 Input Box

Participants used the input box to enter some input into their program and were also asked to modify that input. Below is the question and average score with its responses shown in Figure 5.6.

Q1: Was the Input Box easy to use and manipulate?

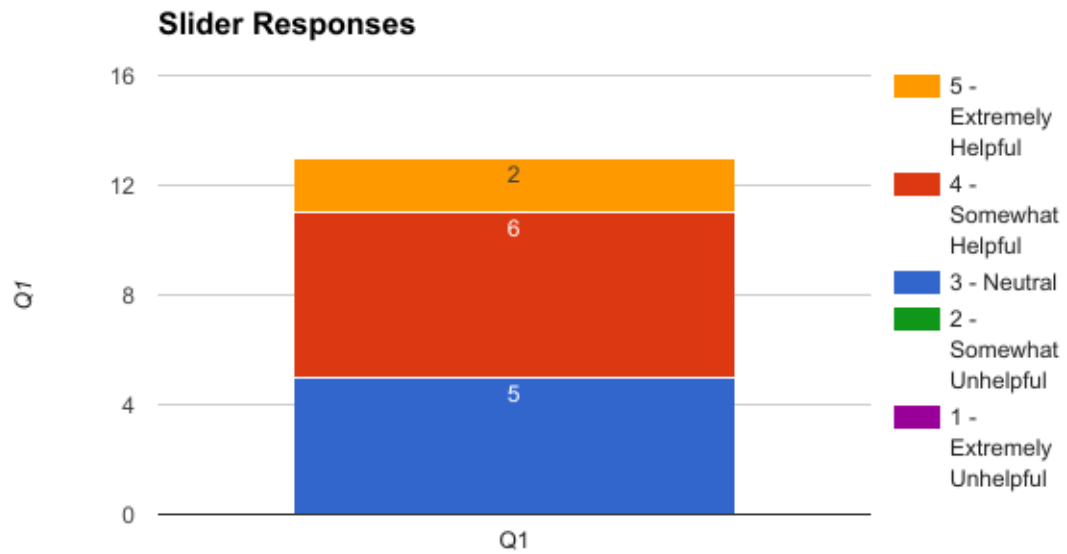


Figure 5.5: Proctored: Sliders Responses

- Average Score: 4.08

The input box is the first feature that one participant found difficult to use with an average score of 4.08. Several participants provided written comments. The comments below expose an error in the environment, provide several suggestions for how to improve the input box, and express their difficulty using the Input Box.

- “Did not show a change when modified the comments”
- “Doesn’t show any errors. For example, if you type in false instead of False or abc instead of ‘abc’ (from the instructions), the output box doesn’t show any of the inputs. Also, it would be cool if the input box showed what input corresponded with what variable.”
- “Unclear as to how it interacts with the program. It would be nice to see “a = ” and then you type in the number for “a”.”

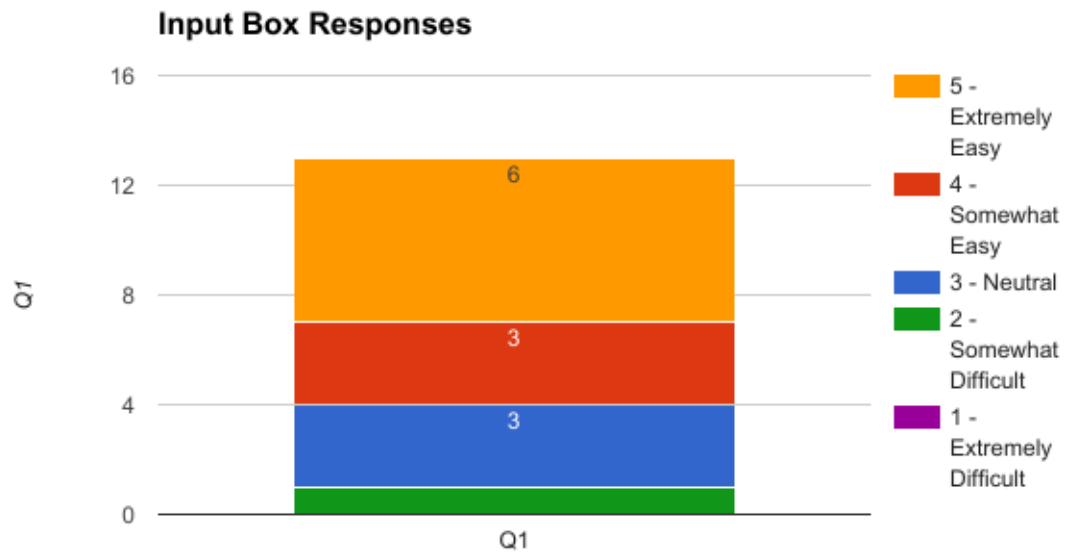


Figure 5.6: Proctored: Input Box Responses

- “Once I understood how to use it, it is an excellent tool. But initially it’s unclear how to use it properly. For example, I thought you had to hit enter if you went back to change an input but apparently this causes some errors.”

In Section 3.2.5, it was noted that a fix was made to the input box to properly display errors for bad input. The above comments are the reason for the fix. One participant also requested that the input box make it clear which variable is being assigned via the use of a label/entry field. This is an interesting suggestion, however making it correct in all cases could prove difficult and would require further investigation.

5.2.6 Output Box

Participants were asked to observe changes in the output box. Below is the list of questions with responses to these questions and average scores shown in Figure 5.7.

Q1: Was the Output Box easy to understand?

- Average Score: 4.54

Q2: Is it easy to understand which Inputs go with which Outputs?

- Average Score: 3.85

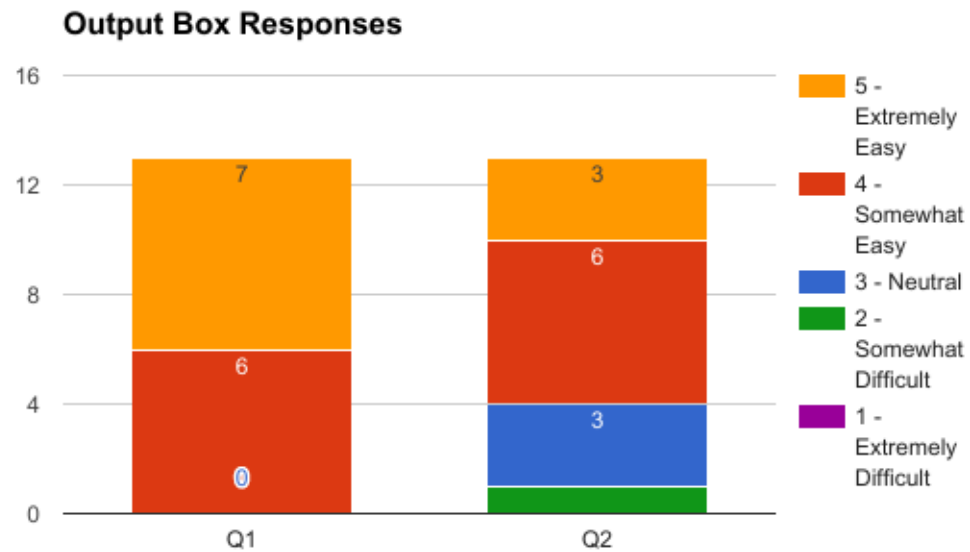


Figure 5.7: Proctored: Output Box Responses

As noted in the responses, average scores, and in the comments below, participants found that in general the output box was easy to understand (Q1: 4.54), but improvements could be made to the output box that focus on helping the participants understand how the input/output box are related and how output is related to the source code (Q2: 3.85).

- “It would be cool to see which variables match with which output. The output box just looks like a list of numbers.”

- “Same as the input box, it would be cool if the output box showed which output corresponded to which variable.”

Adding a feature to the output box that shows which line of code generated that output is a feasible addition that could improve the environment.

5.2.7 Final Comments

Participants were asked to provide any additional feedback at the end of the survey. Some of the response are below:

- “Please send me a copy of this. Please.”
- “If my teacher accepted and acknowledged this, I would much rather use it for my class than anything I have tried up to this point in time. Please get the CSC and CPE departments to use this!”
- “The only suggestion would be to improve the aesthetics but that is not as important as it working. Well done.”
- “I can see myself using this tool, and I can see first time programmers appreciating this tool especially. I would add a tool-tip, if possible, so that users can have an idea of how to use the buttons.”
- “Very cool and works very well. My only issues are with the input and output boxes. They are unclear and I would like to see the actual variable names with their corresponding values in those boxes, otherwise the input/output boxes look like a list of numbers.”
- “This is really cool. I think the idea is awesome and if this takes off I’d definitely use it. Of course, since this is a new project, I don’t expect much in terms of aesthetics. But if user-friendly aspects like that were added, this would be even cooler.”

- “maybe having another window for errors? Or have the output box show errors instead of the variables. I don’t think it really matters, but it kind of makes more sense.”

These response show that many participants found the environment helpful and some have valuable suggestions for how to improve the environment in the future.

5.3 Self Administered Survey Results

The environment was demonstrated in several sections of the introductory computer science courses. The attendees were then asked to complete the online survey. The survey was made available during the 5th week of Spring 2017 and was submitted by 16 participants.

5.3.1 Overall

The participants were asked general questions about their experience with the environment. Below is the list of questions and average scores with the responses to these shown in Figure 5.8.

Q1: Overall did you find the Environment helpful?

- Average Score: 4.38

Q2: Would you use the Environment when coding?

- Average Score: 4.13

Q3: Are the Environment’s aesthetics straightforward? (easy to understand)

- Average Score: 4.13

Q4: Was it easy to use the Environment?

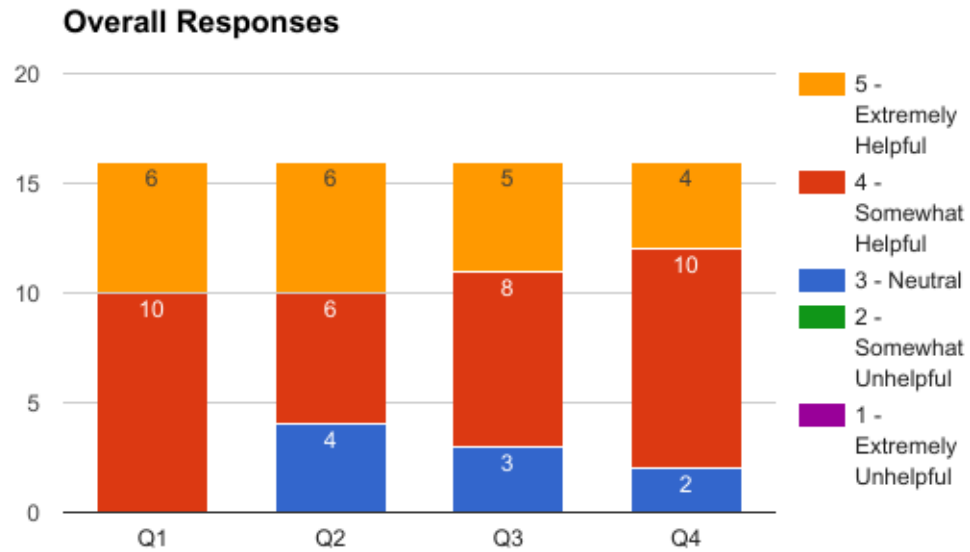


Figure 5.8: Online: Overall Responses

- Average Score: 4.13

From the 16 responses and their average scores, participants were pleased with the environment (Q1: 4.38) and would generally use it when coding (Q2: 4.13). Students found it relatively easy to understand (Q3: 4.13) and use (Q4: 4.13). Some of the comments received from the participants can be found below:

- “I liked the highlight feature incorporated in the Environment. It is a unique design to see the values placed for each line of code. I appreciated how the Environment pin pointed where a syntax or and error occurred within the code to make debugging easier.”
- “Very nice and efficient”
- “The way this breaks down code makes life so much easier.”
- “The environment was easy and simple to use overall”

- “The environment was for the most part easy to use. Although this is for Python 2.7, I wish it was for Python 3.”
- “Good but could use polish”
- “Once I got the hang of using the program, it was almost self-intuitive. However, there definitely is a learning curve.”

The overall feedback from the participants shows that they did indeed find the environment helpful, however programmers may need an introduction on how to use the environment’s features.

5.3.2 Source Code Box

Participants were asked questions about their experience with each of the following features: line highlights, clickable loops/functions, and unit testing a function. Below is the list of questions and average scores with the responses to these questions shown in Figure 5.9.

Q1: Did you find hovering over a line of code and the additional information shown helpful?

- Average Score: 4.69

Q2: For lines of code that called a function (such as ‘add’). Did you find it helpful that the function lines were also highlighted?

- Average Score: 4.25

Q3: Clicking on a Loop: Did you find it helpful to be able to click on a Loop and walk through it?

- Average Score: 4.94

Q4: Clicking on a Function: Did you find it helpful to be able to click on a Function and examine previous calls to that function?

- Average Score: 4.63

Q5: Clicking on a Function: Did you find it helpful to be able to click on a Function and run your own unit test on that function?

- Average Score: 4.94

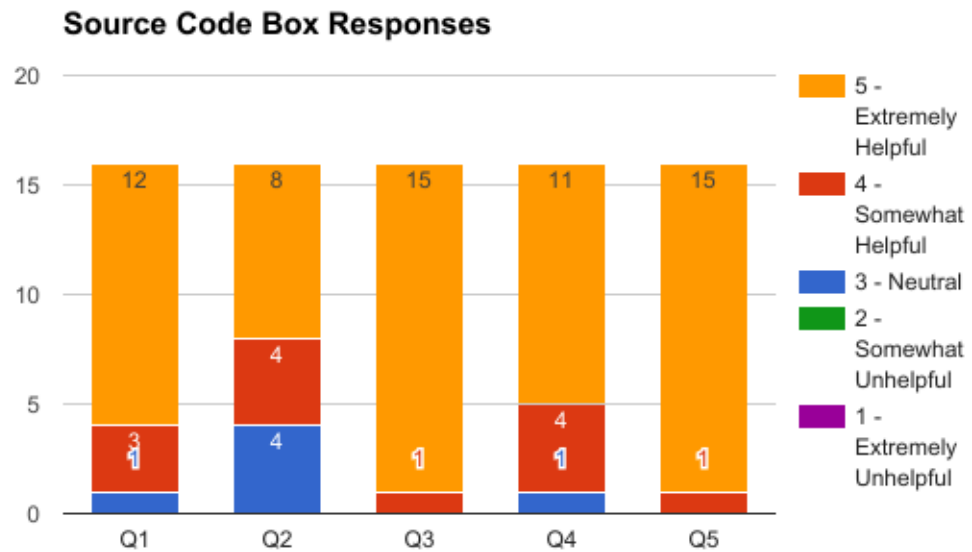


Figure 5.9: Online: Source Code Box Responses

The participants were very pleased with the features added to the source code box, as seen in Figure 5.9, with high average responses for each: Q1 - 4.69, Q2 - 4.25, Q3 - 4.94, Q4 - 4.63, and Q5 - 4.94. Some participant responses for the various features can be found below.

5.3.2.1 Clickable Loop Comments

- “One of the most helpful features”
- “It is nice to be able to see the steps so that it would be easier to troubleshoot.”

- “This was VERY helpful in troubleshooting.”
- “It helped debugging and troubleshooting a ton.”

5.3.2.2 Clickable Function Comments

- “I really liked how it showed the background results”
- “I liked that you could click and see the inner workings of the function”
- “I really like the ability to run tests this quickly.”
- “Yes, it truly broke down each step for the user to visualize”
- “This definitely simplified the entire process of testing functions.”

Overall participants found the source code box and its features helpful. Some participant responses for the source code box overall can be found below.

- “Overall, great idea and great implementation.”
- “Overall, it worked very well for truly understanding/organizing the workings of code.”

5.3.3 Variables Box

Participants were asked about their experience with the variables box. Below is the list of questions and average scores with the responses to these questions shown in Figure 5.10.

Q1: Did you find it helpful to see the state of variables?

- Average Score: 4.5

Q2: Did you find it helpful to have variables separated by their scope? (Global/- Functions)

- Average Score: 4.5

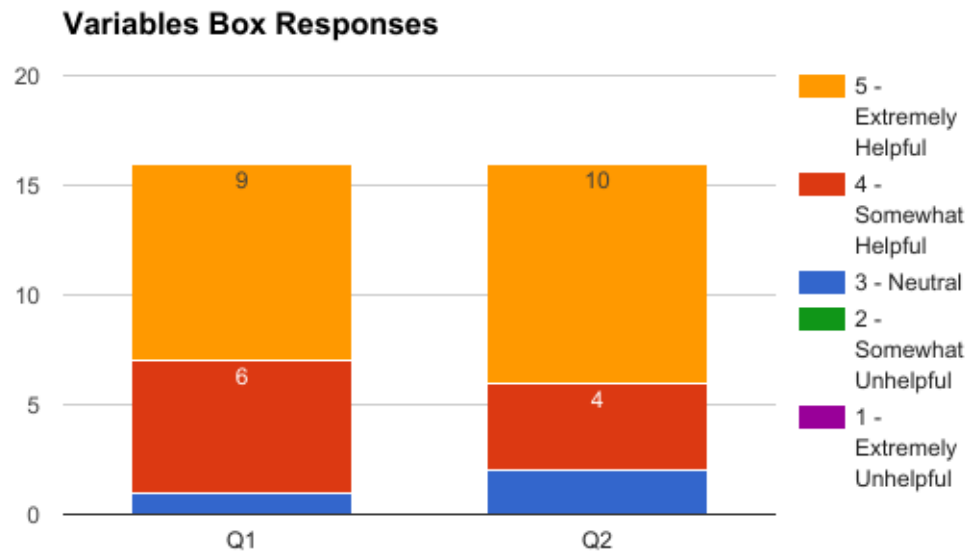


Figure 5.10: Online: Variables Box Responses

The variables box was considered useful by the participants, as seen by the average scores: Q1 - 4.5 and Q2 - 4.5. From the comments below, participants were glad that the variables box kept variables organized and accessible.

- “keeps things organized”
- “Nice to see the difference between what was global and in a function”
- “Makes it easier to keep track of the variables for sure.”
- “Not much to change, each scope separation helped a ton.”
- “Once again, this kept the entire program very organized.”

5.3.4 Changing Program State

Changing program state is when the programmer is able to move forward/backward through their code. The participants were asked to move the slider to see how it would affect the information shown by the environment. Below is the question and

average score with its responses shown in Figure 5.11.

Q1: Did you find it useful to move forward/backward through time via the Sliders and change how much information is shown?

- Average Score: 4.2

Many of the participants found the sliders to be a useful feature in the environment with an average score of 4.2 and shown in the comments below. Though two participants noted that it was difficult and confusing to use.

- “It helped me visualize what aspects of my code the interpreter is going through to calculate such values.”
- “helps you be able to isolate an area and maybe the problem”
- “Maybe it would be helpful if I had more code, but for the limited amount of practice code, it was somewhat unnecessary.”
- “One of the best parts”
- “This ties in to the previous comments; it keeps the entire program extremely organized and easy to understand.”

5.3.5 Input Box

Participants used the input box to enter some input into their program and were also asked to modify that input. Below is the question and average score with its responses shown in Figure 5.12.

Q1: Was the Input Box easy to use and manipulate?

- Average Score: 4.31

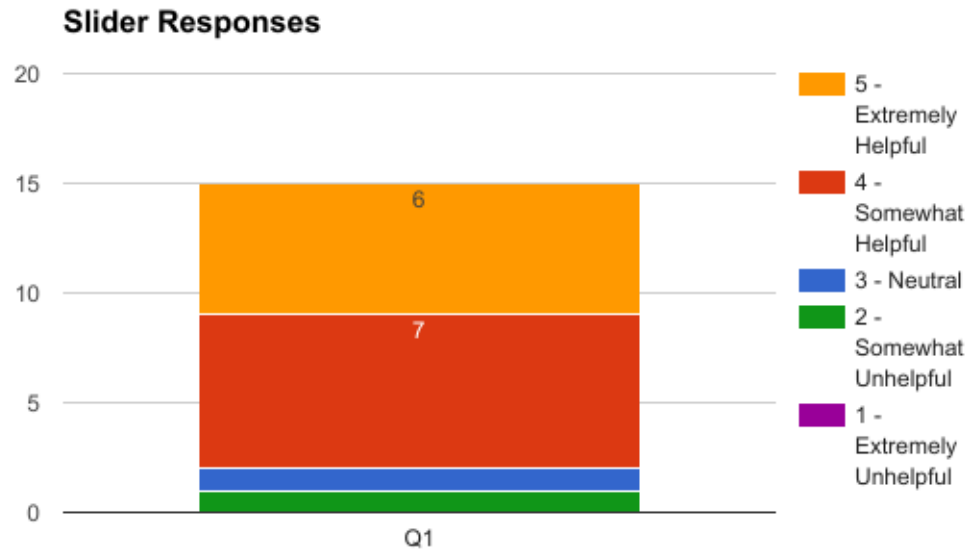


Figure 5.11: Online: Sliders Responses

The input box was well received in the online survey, with an average score of 4.31 and as seen in the comments below. One participant noted some difficulty with the input box, which may have benefited from the changes made in the implementation chapter in section 3.2.5. As discussed in the proctored survey 5.2.5, these changes fixed an error in the environment that notifies the programmer when they enter incorrect input.

- “super helpful”
- “Easy to input”
- “I had trouble with this one”

5.3.6 Output Box

Participants were asked to observe changes in the output box. Below is the list of questions with responses to these questions and average scores shown in Figure 5.13.

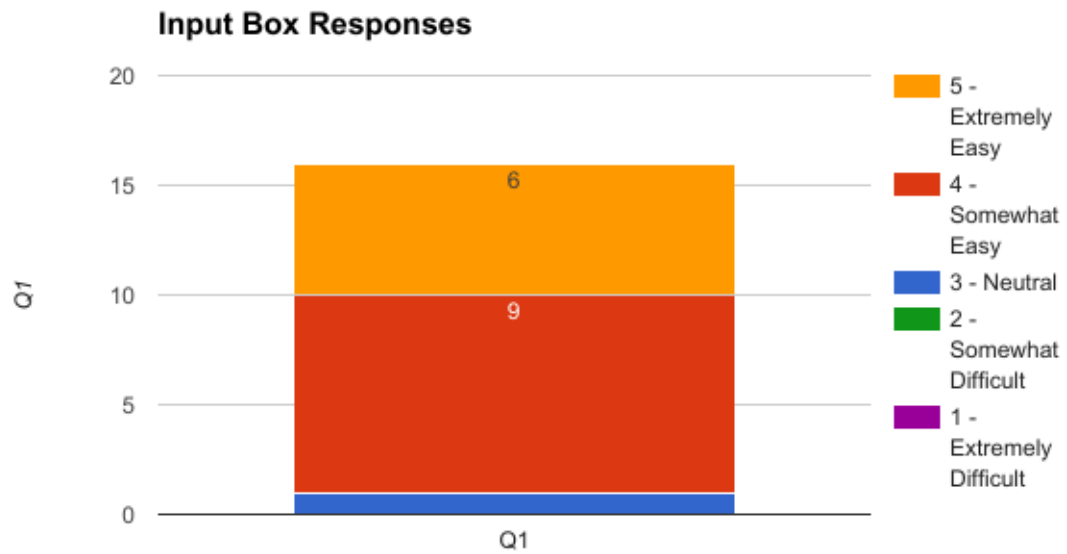


Figure 5.12: Online: Input Box Responses

Q1: Was the Output Box easy to understand?

- Average Score: 4.34

Q2: Is it easy to understand which Inputs go with which Outputs?

- Average Score: 4.13

As noted in the responses, average scores, and in the comments below, participants found that the output box was easy to understand (Q1: 4.34), but generally agreed the output box helps participants understand how the input/output are related and how output is related to the source code (Q2: 4.13).

- “it is good to know what we are actually outputting while we are making code, will make it easier to spot errors”
- “Very easy and simple”
- “I had trouble with this one”

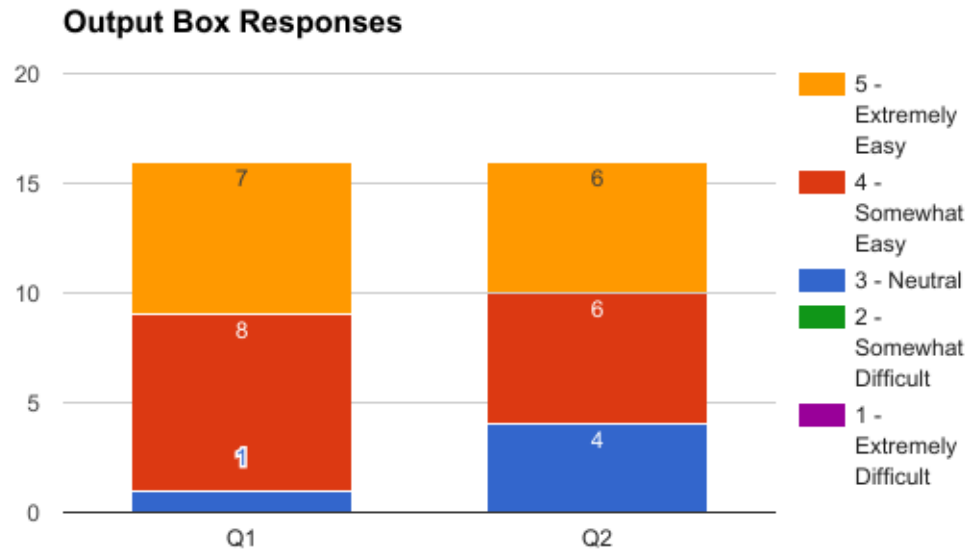


Figure 5.13: Online: Output Box Responses

5.3.7 Final Comments

Participants were asked to provide any additional feedback at the end of the survey. Some of the response are below:

- “Thought it was all around very helpful”
- “It was a little difficult to understand what to do with the input file at first but I figured it out”
- “More directions and possibly a starting guide would be EXTREMELY helpful. Otherwise, once you get the hang of it, the program clarifies coding a TON.”
- “If you could make this available on Windows, that would be extremely helpful for those learning to code. Thanks!”

These response show that some participants found the environment helpful and some have valuable suggestions for how to improve the environment in the future.

5.4 Results

The responses from the participants of the survey provide an overall expression that the live programming environment is beneficial for beginners. While improvements could be made to make the environment aesthetically pleasing, the underlying structure and features in the environment are perceived to benefit the beginner.

One thing to consider when analyzing the results, is how students familiar with PyCharm [6] answered the questions about their experience with the environment developed in this thesis. PyCharm, a very sophisticated IDE, may be overwhelming for a beginner, but the environment in this thesis was designed to be simple and easy to use. One participant noted that “[the environment is] much much easier than clicking back and forth between things like pyCharm and the terminal.” As beginners to programming, participants may not have yet mastered the many features inside of PyCharm and found that the environment in this thesis was able to provide the same information that they were able to attain in PyCharm, but in a more accessible format. Although PyCharm is more powerful, it has a steep learning curve for a beginner. If PyCharm were to integrate the features in this thesis into the IDE, it would be beneficial for beginners as it would simplify the process of understanding their code. Therefore beginners may have found their experience with the environment in this thesis more beneficial than their experience with PyCharm.

Chapter 6

FUTURE WORK

Additional features can be added to improve the environment. Such additions include support for Python 3, a custom python interpreter, multiple source code files, expression granularity, adding better recursion/inner loop support, and addressing feedback received from the survey. Some of these are discussed in more detail below.

6.1 Custom Python Interpreter

A custom Python interpreter would solve many issues that occurred while running the programmer's source code, such as the need for two PDB processes. The goal of the interpreter would be to gather the same information, but without the overhead of a debugger.

6.2 Multiple File Support

Supporting multiple source code files in the environment would allow the programmer to create larger projects. However, it requires a complete overhaul of how information is stored by the environment so that all the features added to the source code box are still supported.

6.3 Expression Granularity

Expression granularity would have allowed the programmer to see a complete breakdown of each line of code. As mentioned, the environment evaluates all expressions in a statement, but only displays the results of each statement. Although

supported structurally within the environment, it is not supported visually in a way that is aesthetically pleasing or accessible to a new programmer. Perhaps having an option to switch between states of granularity would be beneficial.

6.4 Recursion Support

The information for displaying recursive functions and loops within loops is rather poor and requires a lot of rework to be improved so that the environment can properly display the results to the programmer.

6.5 Survey Suggestions

The feedback after performing the survey consists of requests for better aesthetics that look more professional, adding clarity to the input/output box based on the line of source code that they interact with, adding an additional window for errors, and being able to customize the size of the display and hide windows. To improve the aesthetics of the environment, it would be beneficial to use a different GUI framework that uses a more modern set of widgets and enlist the expertise of a graphic designer in HCI and UX.

Chapter 7

CONCLUSION

Learning to program is a difficult process. Beginner's need an environment that will help them learn, develop a mental model of the steps of computation, and learn the syntax of a programming language. The environment proposed and developed in this thesis helps the student understand their code by creating an accessible environment that runs the code and displays the results while the user is typing.

The environment provides many features that help the programmer learn to code. The programmer can highlight individual lines of code to see their results. The programmer can walk through loops and explore the details of a function call. The programmer can unit test a function or class. The environment shows all the variables in the code and creates a visualization for class objects.

The environment appears to be beneficial for beginners based on the two studies performed with a total of 29 participants. The participants were pleased with the many features offered by the environment and they offered critical feedback on how the environment can be improved.

The results of this thesis show that live programming environments should be developed that are targeted towards beginners to make programming more accessible to a larger audience. These environments should help the programmer learn how the code executes and be simple to use.

BIBLIOGRAPHY

- [1] B. Fry and C. Reas. Processing. <https://processing.org/>. [Online: accessed 7-31-2017].
- [2] P. Guo. Python tutor. <http://www.pythontutor.com/>, 2013. [Online: accessed 7-31-2017].
- [3] C. M. Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Cambridge, MA, USA, 2003. AAI0805688.
- [4] HARC. Human advancement research community. <https://harc.ycr.org/>, 2016. [Online: accessed 7-31-2017].
- [5] IRB. Institutional review board. <http://research.calpoly.edu/HS-policy>. [Online: accessed 7-31-2017].
- [6] JetBrains. Pycharm. <https://www.jetbrains.com/pycharm/>. [Online: accessed 7-31-2017].
- [7] Kodowa. Light table. <http://lighttable.com/>. [Online: accessed 7-31-2017].
- [8] M. Lutz. *Programming Python, 2nd Edition*, chapter 17.10. O'Reilly Media, 2001.
- [9] S. McDirmid. A live programming experience. <https://www.youtube.com/watch?v=YLrdhFEAiQo>. [Online: accessed 7-31-2017].
- [10] S. McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on*

Programming & Software, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM.

- [11] Pygments. Pygments. <http://pygments.org/faq/>. [Online: accessed 7-31-2017].
- [12] Python. Ast node visitor class.
<https://docs.python.org/2/library/ast.html#ast.NodeTransformer>.
[Online: accessed 7-31-2017].
- [13] Python. The python debugger.
<https://docs.python.org/2/library/pdb.html>. [Online: accessed 7-31-2017].
- [14] Python. Python for beginners.
<https://docs.python.org/3/faq/general.html#is-python-a-good-language-for-beginning-programmers>. [Online: accessed 7-31-2017].
- [15] Python. Python’s abstract syntax tree.
<https://docs.python.org/2.7/library/ast.html>. [Online: accessed 7-31-2017].
- [16] Python. Tkinter. <https://docs.python.org/2/library/tkinter.html>.
[Online: accessed 7-31-2017].
- [17] B. Victor. Bio. <http://worrydream.com/Bio/>. [Online: accessed 7-31-2017].
- [18] B. Victor. Up and down the ladder of abstraction.
<http://worrydream.com/LadderOfAbstraction>, 2011. [Online: accessed 7-31-2017].
- [19] B. Victor. Learnable programming.

<http://worrydream.com/LearnableProgramming>, 2012. [Online: accessed 7-31-2017].

APPENDICES

Appendix A

INFORMED CONSENT FORM

Survey: A Live Programming Environment for Python by Noah Negrey

Program: Loops, Functions, Input, & Output

* Required

Informed Consent Form

INFORMED CONSENT TO PARTICIPATE IN A RESEARCH PROJECT, "A Live Programming Environment for Python"

A research project on a live programming environment for python is being conducted by Noah Negrey, a graduate student in the Department of Computer Science at Cal Poly, San Luis Obispo. The purpose of the study is to determine if students find the environment useful when learning to program. This also means determining what students like or dislike about the tool as well as asking for their input on what might be missing from the tool.

You are being asked to take part in this study by completing the following questionnaire. You will be asked to follow a guided programming exercise, in which you will interact with different features in the tool and then asked questions about your experience with the tool. Your participation will take approximately 45 minutes. Please be aware that you are not required to participate in this research, you may omit any items that you prefer not to answer, and you may discontinue your participation at any time without penalty.

Your responses will be provided anonymously to protect your privacy.

If you have questions regarding this study or would like to be informed of the results when the study is completed, please feel free to contact Noah Negrey at nnegrey@calpoly.edu. If you have concerns regarding the manner in which the study is conducted, you may contact Dr. Michael Black, Chair of the Cal Poly Institutional Review Board, at (805) 756-2894, mblack@calpoly.edu, or Dr. Dean Wendt, Dean of Research, at (805) 756-1508, dwendt@calpoly.edu.

If you agree to voluntarily participate in this research project as described, please indicate your agreement by completing and submitting the following questionnaire. Please print a copy of this consent form now for your reference, and thank you for your participation in this research. *

☐ Yes, I Volunteer

NEXT

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. [Report Abuse](#) - [Terms of Service](#) - [Additional Terms](#)

Google Forms



Appendix B

ENVIRONMENT OVERVIEW

Survey: A Live Programming Environment for Python by Noah Negrey

Loops & Functions Lab

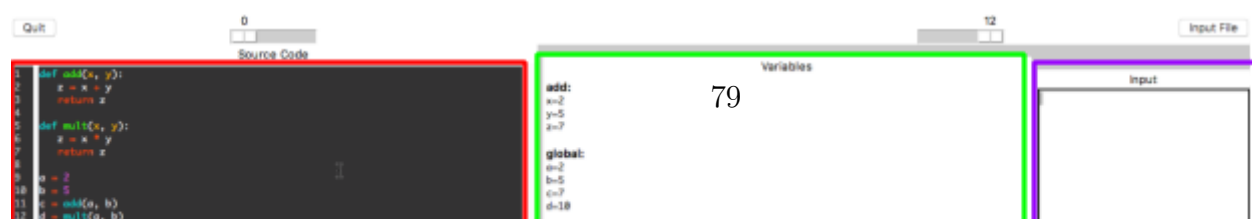
Please open the Google Document and follow the steps provided before moving on to the survey. See below for an Introduction to the Environment.

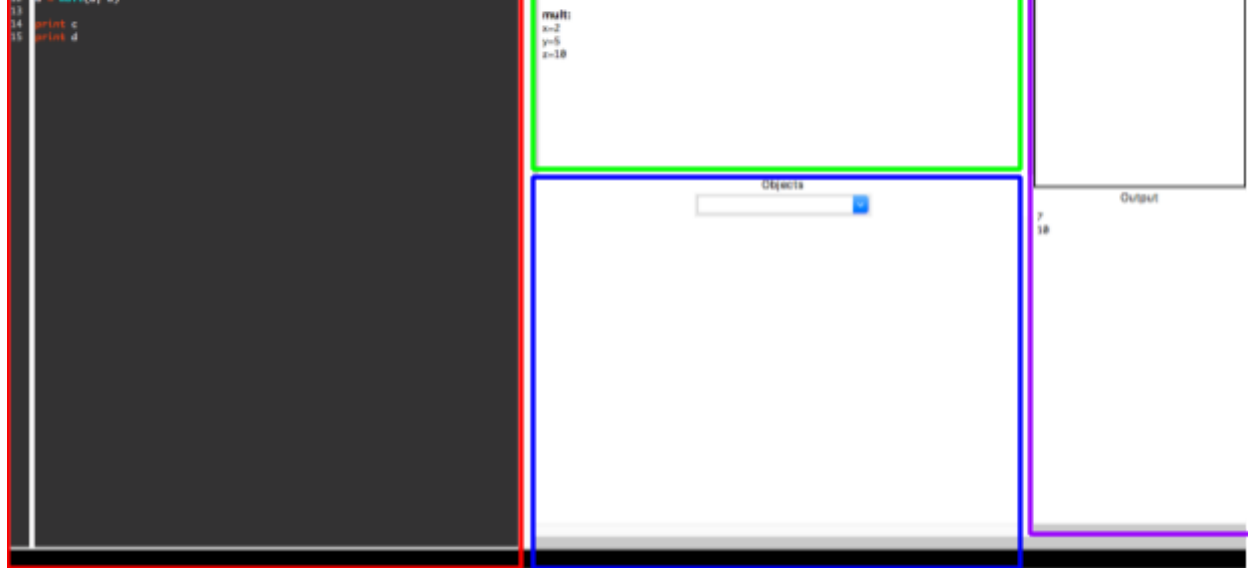
https://docs.google.com/document/d/18ot7vWbY1a77gSr7CYHAayzdSQSrqeCC-N_X0Z_3Rxo/edit?usp=sharing

Environment Overview

The Environment, as seen in Figure 1, has an example program that shows you some aspects of the tool.

Figure 1





Source Code Box (Red)

The area highlighted in Red is for the user's source code and where the user will mostly interact. As seen it does some syntax highlighting and a few other features that the user will explore during the experiment.

Variables Box (Green)

The Green box keeps track of the programs variables and the state in which they occur.

Objects Box (Blue)

The Blue box creates a graphical representation for classes/objects that the user creates, such as Trees or Lists.

Input/Output Boxes (Purple)

Lastly, the Purple box has an Input section above for any input that is needed for the code and an output section below to display any output from the user's program.

BACK

NEXT

Never submit passwords through Google Forms.

Appendix C

SURVEY SETUP

Setup

If you are a Cal Poly student using a lab machine there is a script that will open the environment for you.

If you are a Non Cal Poly student or off campus, you will have to download the source code and run the program from the command line.

Cal Poly Lab Machine

- Option 1: Run the following command from your terminal:
 - `/home/nnegrey/Public/LiveProgrammingEnvironment.sh`
- Option 2: Download and run this [Script](#)
 - You'll need to call this on the script.
`chmod 700 LiveProgrammingEnvironment.sh`

Note: I've been informed that there might be some Python Path issues with the Pygments Module in my directory. If that is the case you can install it locally with this call:

```
pip install --user pygments
```

Source

Direct Source (Cal Poly Servers)

Download all the source files found at [nnegrey's folder](#) to a local directory.

Navigate to the source code via the command line.

Run

```
python live.py
```

You may need to install the Pygments Dependency

```
pip install pygments
```

or

```
pip install --user pygments
```

or

- Build Pygments from source found [here](#)
- Navigate to the downloaded directory.

```
python setup.py build
```

```
python setup.py install --user
```

Appendix D

PROGRAM: LOOPS, FUNCTIONS, INPUT, & OUTPUT

Survey Program: A Live Programming Environment for Python by Noah Negrey

Loops, Functions, Input, & Output

Explanatory (plain text)

Instructions (bold text)

Task 1: Setup

Please follow the steps found in this [link](#) to setup the environment.

Once the environment is up and running, it should look something like Figure 1 below.

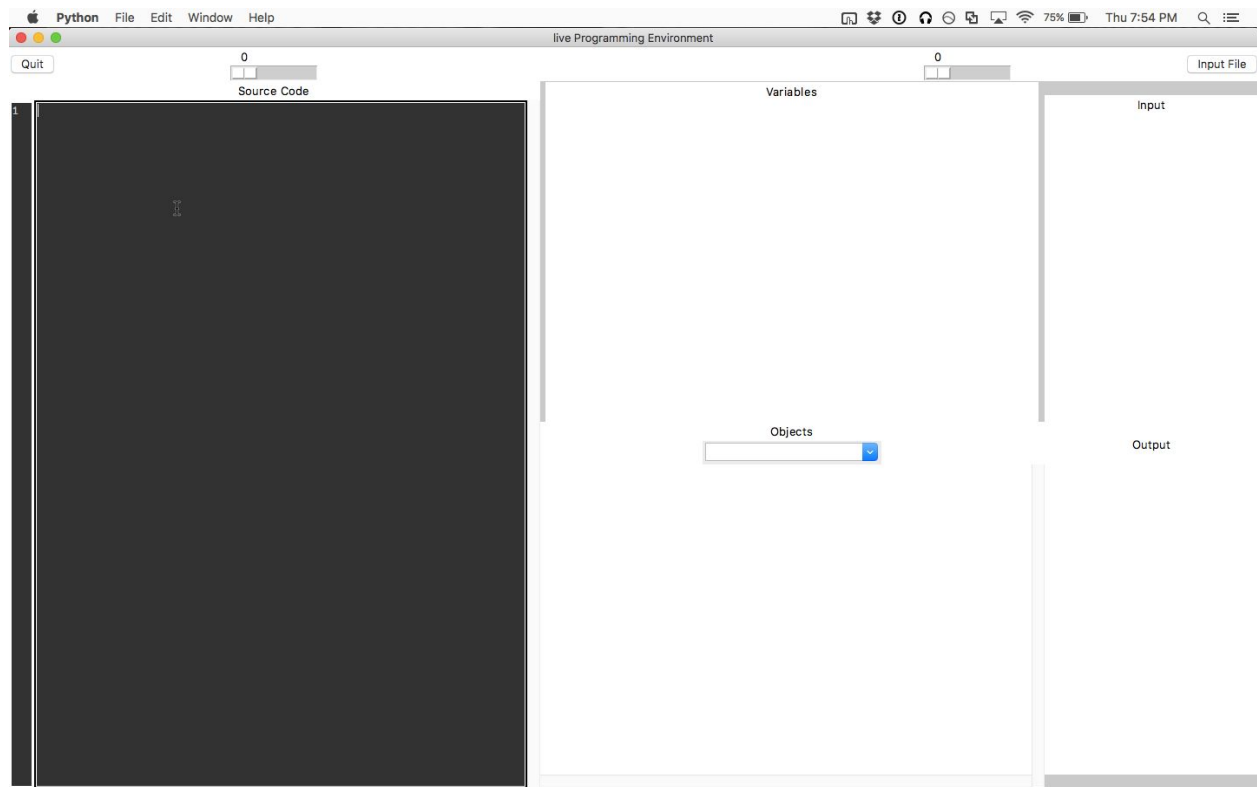


Figure 1: The Environment when first opened

Task 2: Environment Exploration

After that we'll begin entering some code into the box labeled "Source Code" (it is also the dark gray box).

(While typing this you may see a red highlight over the line, we'll ignore this for now)

Task 2.1: The Variables Box

Insert:

```
a = 1
```

After typing this, you should see something change in the Variables box. Such as:

```
global:  
a=1
```

This is information about the variable 'a' in our program and its current value, future variables will be declared here. If you change the value of 'a' the value in the variables box will also be updated.

Task 2.2: Line Highlights

The next part of the environment that can be explored is line highlights. Line highlights occur when the mouse cursor is over a line of code, it highlights the selected line and display the resulting value for that line.

Insert a new line:

```
b = 1 + 1
```

After typing this, move your mouse over the line and you should see something like this:

```
b = 1 + 1      2
```

Where the 2 is the resulting value for that line.

Task 3: Loop Exploration

Task 3.1: Simple Loop

For Task 3 we will create a simple loop that will repeat 5 times and print values to the screen. To do this enter the code below:

Insert:

```
i = 0  
while i < 5:  
    print i  
    i += 1
```

After typing this you should see in the variable box that i is now equal to **5** and in the output box you should see something like this:

```
0  
1
```

2
3
4

Highlight over the different lines in the loop.

You can see the resulting value for each line, the line with the while loop will have the value True, since it is a conditional statement.

Task 3.2: Walking through a Loop

Click on the line that has the 'while' statement:

`while i < 5:` (Click on this line)

After clicking on that line you should see a popup box like the one in Figure 2.

The box has three main parts to it. The slider value, the text on the left, and the text on the right.

- The text on the left is the same as the code inside the looping function. (This will not change)
- The text on the right is the evaluation of that line of code.
- The slider value contains all iterations of the loop

As you move the slider along, it will change the text on the right side to show you all the information in that loop. It ends when the conditional is no longer true.

Move the slider to the value 3, the right hand side should look like this:

True
3
i=4

Close the window when finished.

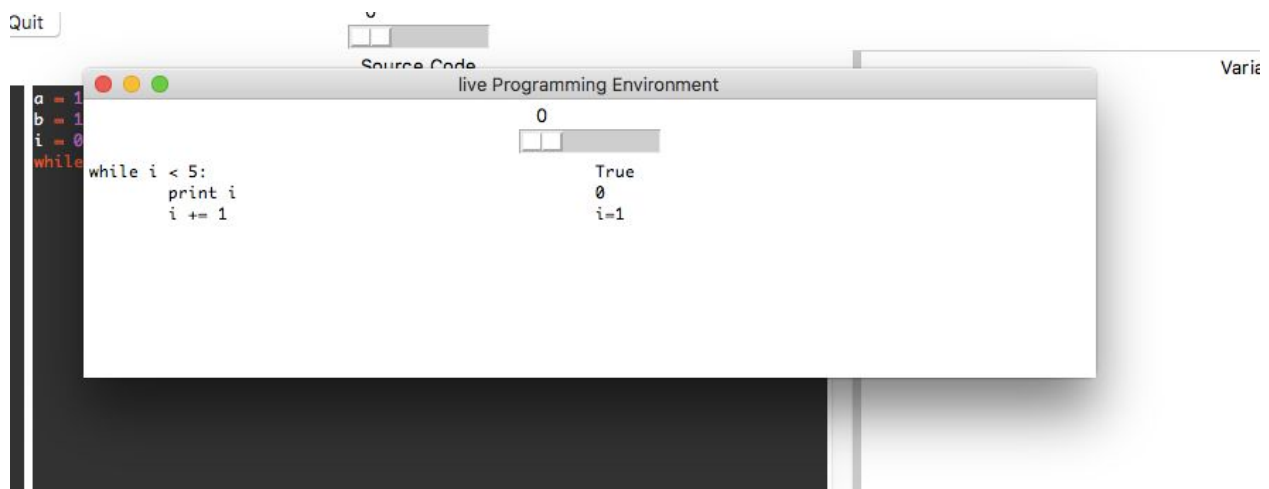


Figure 2: Walking through a loop popup box.

Task 4: Function Exploration

Task 4.1: Simple Function

We'll write a simple add function that we will call twice and print the results to the screen.

Insert:

```
def add(x, y):  
    z = x + y  
    return z  
  
c = add(a, b)  
d = add(i, i)  
print c  
print d
```

After typing this, you should see two new outputs in the Output box '3' and '10'. In the Variable box you should see a new set of variables for the function state.

```
add:  
x = 5  
y = 5  
z = 10
```

These values correspond to the second call that we made to the add function.

Task 4.2: Exploring Function Highlights

For this step we'll be doing the same thing that we did in Task 2.2, but for these two lines:

Highlight over this line in your code.

```
c = add(a, b)
```

For the line: `c = add(a, b)`, it should look like this:

```
def add(x, y):          y=2, x=1  
    z = x + y          3  
    return z           3  
  
c = add(a, b)          3
```

Highlight over this line in your code.

```
d = add(i, i)
```

For the line: `d = add(i, i)`, it should look like this:

```
def add(x, y):          y=5, x=5
```

```

z = x + y      10
return z      10

c = add(i, i)   10

```

You'll notice that on top of having the selected line's information appear, so does the called function. It shows any parameters to that function and the resulting values for the lines of code in that function.

Task 4.3: Exploring Function Calls

Click on the line that declares the 'add' function:

```
def add(x, y):      (click on this line)
```

After clicking on that line you should see a popup box like the one in Figure 3.

The box has 5 main parts to it.

- The drop down menu: this contains all the places that this function is called in your code
- The 'Test Function' Button: We'll discuss this in Task 4.4
- The left side with text: is the same as the code inside the function. (This will not change)
- The right side (currently without text): Will change based on your selection from the drop down menu.
- The bottom function parameter section: We'll discuss this in Task 4.4

In the drop down menu, there should currently be two selections. They should look like this:

Line #: y=2, x=1

Line #: y=5, x=5

Select the first option and it will show you the execution of your function for that selected call.

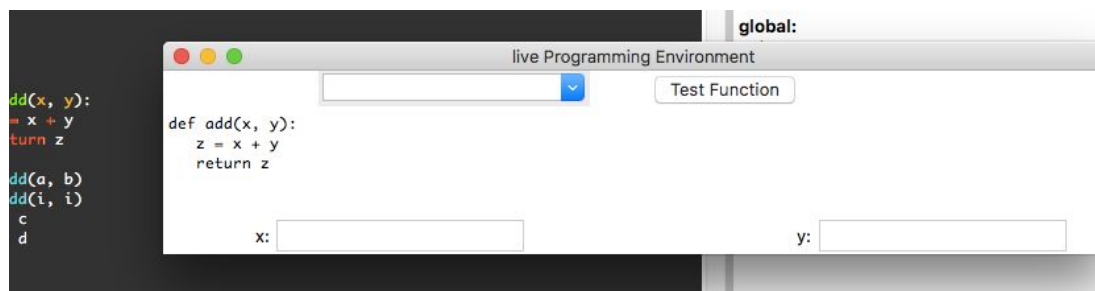


Figure 3: Popup Box after clicking on a function line

Task 4.4: Testing a Function

Now we'll be looking at how to test the function quickly with different inputs.

To do this we'll first discuss the two parts that we have not yet introduced.

Test Function Button: The test function button, will execute the function code with the specified input.

Function Parameter Section: In this section the popup box will create inputs for all the parameters to the function.

In the 'x' field, enter: 7

In the 'y' field, enter: 8

Then hit the 'Test Function' button.

You should see this in the right side box:

```
y=8, x=7
z=15
15
```

This allows you to quickly test a function, but now we'll test the function with some different inputs:

In the 'x' field, enter: abc

In the 'y' field, enter: xyz

Then hit the 'Test Function' button.

Now you should see this in the right side box:

```
y='xyz', x='abc'
z='abcxyz'
'abcxyz'
```

Close the window when finished.

Task 5: Changing Program State

The last thing you'll do is examine the top right slider. This slider controls how much information is shown. It can be helpful to step through your program to determine what each variable is in the Variables box and walk through the program as output appears in the Output box.

Move the slider to 10.

You'll notice that the Variable box now only has three variables and that 'i' is equal to 2. While the Output box only has the output for '0', '1', and '2'.

Task 6: Adding Some Input & Output

To start we'll add a few very simple input output lines of code to our program.

Note: It may appear that the program stops working, because it is waiting for input, which will be added in the next step.

Enter:

```
a = input()
b = input()
c = input()
d = input()
```

```
print a
print b
print c
print d
```

After typing in that information, the program won't do anything until you provide it with some input.

Select the Input box (on the top right side of the environment)

Enter in the Input Box:

```
1
2
3
4          (Be sure to hit 'enter' after this line)
```

(Make sure to have an empty fifth line here, this acts as you hitting enter on the terminal)

Now you should see all four variables (a, b, c, d) in the Variable box change based on the input and the Output box will also have four new lines.

Task 7: Changing Input

In the Input Box, change the input 2 to False

In the Input Box, change the input 3 to 'abc'

After typing these in, notice that the Variable and Output box change for the new input.

Task 8: Using an Input File

Open up a new file and add these four things to the file with a new line after each one.

Enter:

```
4
3
2
1          (Be sure to hit 'enter' after this line)
```

(Make sure to have an empty fifth line here, this acts as you hitting enter on the terminal)

Save the file as 'input.txt'

Open back up the Environment.

In the top right corner select the button 'Input File'

This will open up a file explorer for you to use to find the 'input.txt' file.

Select the file to see the Input, Variable, and Output boxes change.

Task 9: Return to the Survey

Appendix E

PROGRAM SURVEY: LOOPS, FUNCTIONS, INPUT, & OUTPUT SURVEY

Survey: A Live Programming Environment for Python by Noah Negrey

Loops & Functions Survey

The goal of this survey is to learn about your experience with the Live Programming Environment and discover what features you liked and disliked.

Overall

These questions are used to measure your overall experience with the Environment

Overall did you find the Environment helpful?

- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

Would you use the Environment when coding?

- ☐ 5 - Extremely Likely
- ☐ 4 - Somewhat Likely
- ☐ 3 - Neutral



☐ 2 - Somewhat Unlikely

☐ 1 - Extremely Unlikely

Are the Environment's aesthetics straightforward? (easy to understand)

☐ 5 - Extremely Straightforward

☐ 4 - Somewhat Straightforward

☐ 3 - Neutral

☐ 2 - Somewhat Confusing

☐ 1 - Extremely Confusing

Was it easy to use the Environment?

☐ 5 - Extremely Easy

☐ 4 - Somewhat Easy

☐ 3 - Neutral

☐ 2 - Somewhat Difficult

☐ 1 - Extremely Difficult

Overall Environment Comments

Your answer

Source Code Box

These next questions are targeted toward the source code box

Did you find hovering over a line of code and the additional information shown helpful?

Quit

94

Source Code

```
1 a = 1
2 b = 1 + 1 2
```



```
3 i = 0
4 while i < 5:
5     print i
6     i += 1
7
8 def add(x, y):
9     z = x + y
10    return z
11
12 c = add(a, b)
13 d = add(i, i)
14 print c
15 print d
```

- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

For lines of code that called a function (such as 'add'). Did you find it helpful that the function lines were also highlighted?

Quit

0 95

Source Code

1 a = 1

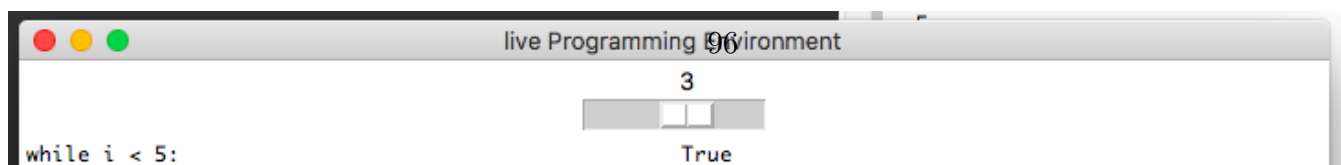
```

2 b = 1 + 1
3 i = 0
4 while i < 5:
5     print i
6     i += 1
7
8 def add(x, y):      y=5, x=5
9     z = x + y      10
10    return z        10
11
12 c = add(a, b)
13 d = add(i, i)      10
14 print c
15 print d

```

- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

Clicking on a Loop: Did you find it helpful to be able to click on a Loop and walk through it?



```
print i
i += 1
```

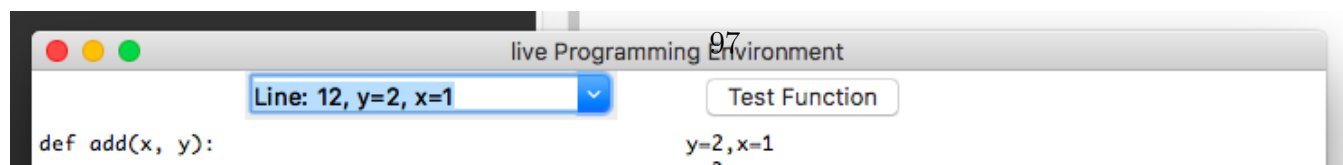
```
3
i=4
```

- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

Clickable Loop Comments

Your answer

Clicking on a Function: Did you find it helpful to be able to click on a Function and examine previous calls to that function?



y:

- ☐ 1 - Extremely Unhelpful

Clicking on a Function: Did you find it helpful to be able to click on a Function and run your own unit test on that function?

y: xyz

- ☐ 1 - Extremely Unhelpful

Clickable Functions Comments

Your answer

Source Code Box Comments

Your answer



These next questions are targeted toward the Variables box

Did you find it helpful to see the state of variables?

```
Variables
add:
x=5
y=5
z=10
global:
a=1
b=2
i=5
c=3
d=10
```

- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

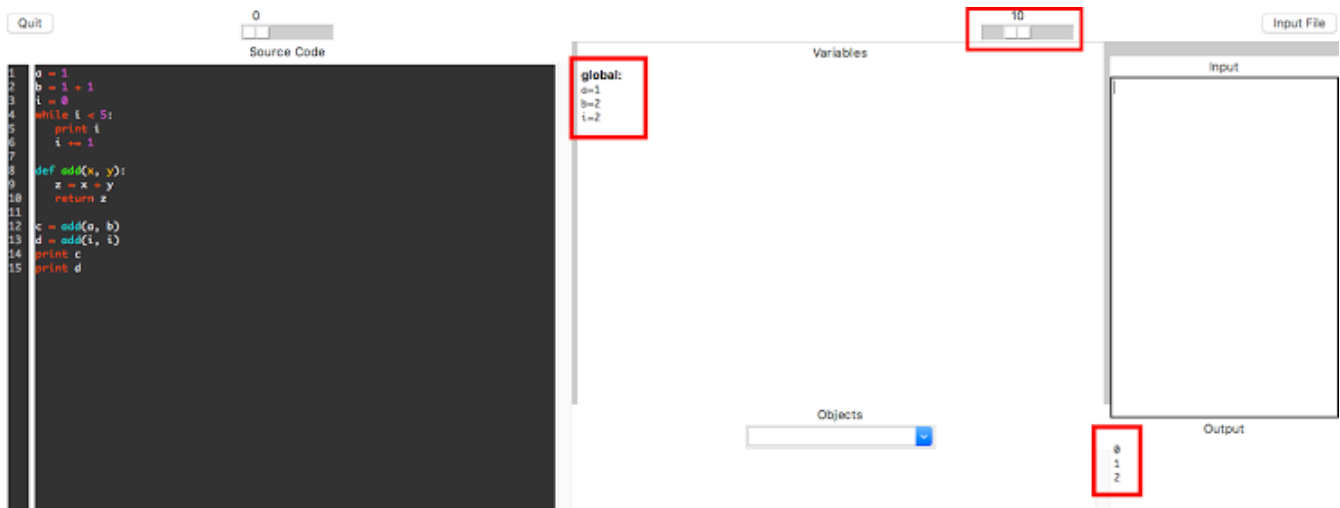
Did you find it helpful to have variables separated by their scope?
(Global/Functions)

- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

Variables Box Comments

Your answer

Did you find it useful to move forward/backward through time via the Sliders and change how much information is shown?



- ☐ 5 - Extremely Helpful
- ☐ 4 - Somewhat Helpful
- ☐ 3 - Neutral
- ☐ 2 - Somewhat Unhelpful
- ☐ 1 - Extremely Unhelpful

Sliders Comments

Your answer

Input Box

These next questions are targeted toward the Input box

Was the Input Box easy to use and manipulate?

- ☐ 5 - Extremely Easy
- ☐ 4 - Somewhat Easy
- ☐ 3 - Neutral

☐ 2 - Somewhat Difficult

☐ 1 - Extremely Difficult

Input Box Comments

Your answer

Output Box

These next questions are targeted toward the Output box

Was the Output Box easy to understand?

☐ 5 - Extremely Easy

☐ 4 - Somewhat Easy

☐ 3 - Neutral

☐ 2 - Somewhat Difficult

☐ 1 - Extremely Difficult

Is it easy to understand which Inputs go with which Outputs?

☐ 5 - Extremely Easy

☐ 4 - Somewhat Easy

☐ 3 - Neutral

☐ 2 - Somewhat Difficult

☐ 1 - Extremely Difficult

Output Box Comments

Your answer

Do you have any remaining comments? (Successes, Frustrations, Suggestions, Criticisms).

Your answer

BACK

SUBMIT

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. [Report Abuse](#) - [Terms of Service](#) - [Additional Terms](#)

Google Forms

