Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations

Hyeonsu Kang
UC San Diego
La Jolla, CA, USA
hyk149@eng.ucsd.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

Visualizations of run-time program state help novices form proper mental models and debug their code. We push this technique to the extreme by posing the following question: *What if a live programming environment for an imperative language always displays the entire history of all run-time values for all program variables all the time?* To explore this question, we built a prototype live IDE called Omnicode ("Omniscient Code") that continually runs the user's Python code and uses a scatterplot matrix to visualize the entire history of all of its numerical values, along with meaningful numbers derived from other data types. To filter the visualizations and hone in on specific points of interest, the user can brush and link over the scatterplots or select portions of code. They can also zoom in to view detailed stack and heap visualizations at each execution step. An exploratory study on 10 novice programmers discovered that they found Omnicode to be useful for debugging, forming mental models, explaining their code to others, and discovering moments of serendipity that would not have been likely within an ordinary IDE.

Author Keywords

live programming; always-on visualizations

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

One of the most fundamental skills that students learning programming need to master is being able to form mental models of how static pieces of source code correspond to dynamic run-time actions inside of a computer [10, 27]. Without this basic core foundation, it is impossible to become a proficient programmer. For instance, consider this tiny Python example: `x=[1,2,3]; y=x; x[0]=100`. What is the value of `y[0]` after this code runs? To be able to answer even this simple question, a student must develop a viable mental model of how the Python = operator affects lists and integers.

UIST 2017, October 22–25, 2017, Quebec City, QC, Canada

ACM 978-1-4503-4981-9/17/10...$15.00
https://doi.org/10.1145/3126594.3126632

Currently, the most widely-used tools for helping novices develop this critical skill are either too primitive or too hard to use. Perhaps the most ubiquitous tool here is the humble print statement: Littering print statements throughout one's code – often toggled by commenting and uncommenting lines – is a low-tech, messy, but somewhat reasonable way to probe run-time values. Symbolic debuggers are more elegant and powerful, but those can be hard for novices to use since they must learn how to set breakpoints and watchpoints. Unlike print statements, debuggers do not provide a holistic overview of how run-time values change throughout program execution.

Live programming environments [1, 5, 6, 20, 30, 32] improve upon print statements and debuggers by continually running code and displaying its run-time values. However, existing live environments for general-purpose imperative languages (e.g., Java [5], JavaScript [20], Python [13]) display only the most recent value for each executed line, expression, or active object – not the entire history of all values. In this paper, we want to push this promising idea of liveness to one logical extreme by asking: *What if a live programming environment for an imperative language always displays the entire history of all run-time values for all program variables all the time?*

At first glance, this idea seems impractical. How could it ever scale? Our key insight is that it does *not*, in fact, need to scale. Our target audience is novices learning to solve algorithmic problems by writing small, self-contained pieces of code. As a ballpark estimate, this kind of code contains around 10 variables and runs for around 100 execution steps. Displaying the values of 10 variables at all 100 steps is only 1,000 data points, which is well within reach of data visualization tools.

Despite being small in size, this kind of self-contained algorithmically-focused code is pervasive: Millions of students who are learning programming in both residential classes and MOOCs all need to write this type of code for their assignments. In addition, everyone who is preparing for software engineering and data science technical job interviews needs to practice solving algorithmic coding problems; their solution code must usually be small enough to fit on a whiteboard during interviews. Also, everyone who enters programming contests must also train on similar problems.

To help these millions of novices develop mental models for the kind of code they write for class or interview prep, we created *Omnicode* ("Omniscient Code"), a live IDE with always-on run-time visualizations. Figure 1 shows a usage scenario:

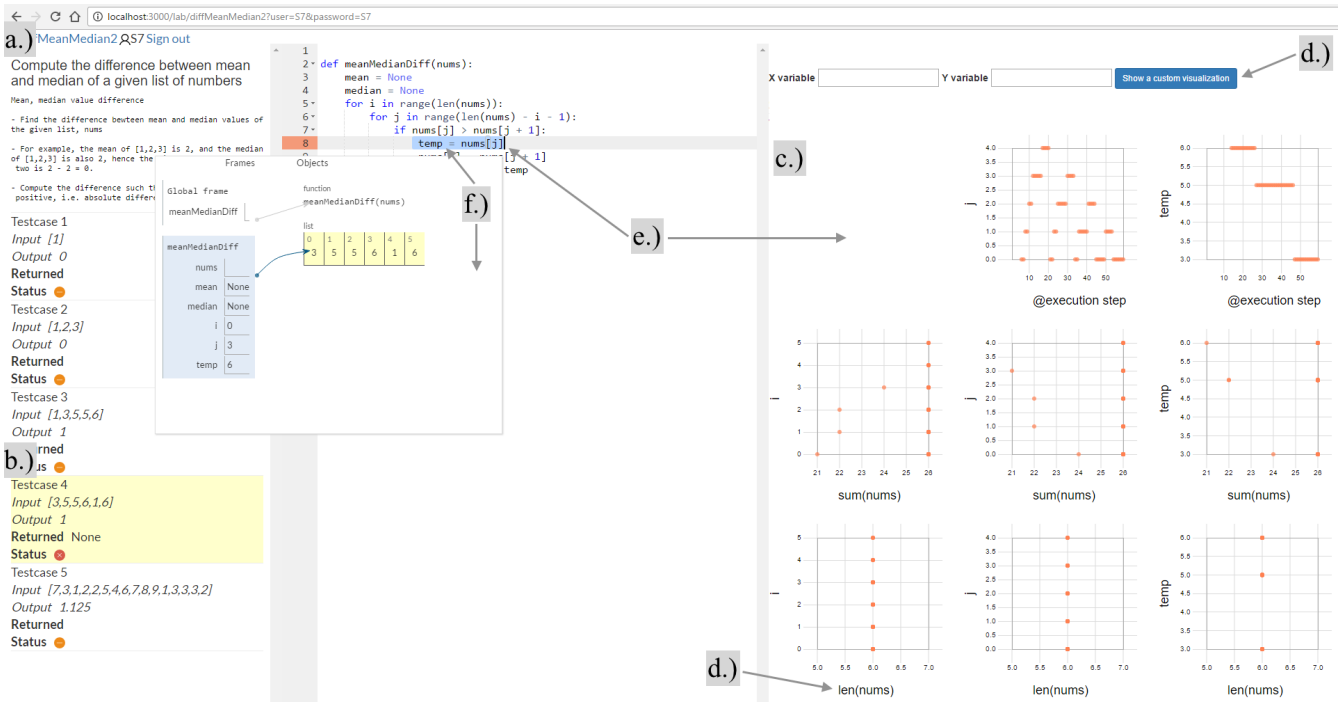

Figure 1. Omnicode is an IDE for novice programmers that allows the user to: a.) load a programming problem and test cases, b.) write Python code and run test cases, c.) see live visualizations of run-time values on a scatterplot matrix, d.) visualize derived values and arbitrary Python expressions, e.) filter the visualizations by brushing over either the code or the scatterplots, f.) zoom in to view all variables and data structures at an execution step.

(a) The user starts with a programming problem statement, a set of test cases, and a code editor seeded with starter code.

(b) As the user is coding, Omnicode continuously runs their code, reports test status and error messages in the left pane, and displays the values of all program variables in the right pane. There is no "Run" button; everything updates live.

(c) The run-time value visualization is a matrix of scatterplots. Each numerical value is plotted both relative to all execution steps (to show the entire history of how it changed over time) as well as relative to all other values (to show dyadic two-variable correlations).

(d) Omnicode automatically derives relevant numerical values for non-numeric types such as the lengths of lists and strings. The user can also write arbitrary Python expressions to visualize on the scatterplots, such as `sum(x)` or `pow(x, 2)` for the sum of list elements or x^2, respectively.

(e) To counteract visual overload, Omnicode implements *bi-directional brushing-and-linking*: The user can select a region of source code (e.g., a variable or set of lines) and the scatterplot matrix gets filtered to include only data points relevant to the user's code selection. Conversely, the user can select any region of any scatterplot, and all other scatterplots as well as the source code selection get updated to include only data that is relevant to the user's selection.

(f) Finally, to probe run-time state in more detail, the user can select a line of code and see a pop-up tooltip that shows a complete heap visualization of all data structures present when that line executes (powered by Python Tutor [13]). Whereas the scatterplot matrix shows only numbers, this inline visualization shows values of *all* Python data types.

To our knowledge, Omnicode is the first attempt to display always-on live visualizations of the entire history of all numerical program values across all execution steps. Although Omnicode visualizes only numerical values, the type of algorithmic student code that it targets most frequently manipulates numbers; in addition, users can visualize arbitrary number-producing expressions (Figure 1d) and call on Python Tutor to visualize all Python data types (Figure 1f).

To investigate whether Omnicode can help novices debug, form mental models, and explain how their code works to others, we ran an exploratory user study on 10 university students who identified as novice programmers. Subjects used Omnicode to solve three programming problems of the sort seen in introductory courses and coding interviews. They found Omnicode useful as both a helper for forming mental models and as a visual explanatory aid for teaching code execution semantics. Some even found serendipity by glancing at the always-on visualizations and embarked on impromptu explorations that they would not have done within a normal IDE.

The contributions of this paper are:

- The idea of pushing live programming for general-purpose languages to the extreme by displaying the entire history of all run-time values for all program variables all the time.
- Omnicode, a prototype live IDE that takes steps toward this idea using always-on, bidirectionally-linked visualizations.
- An exploratory user study demonstrating that Omnicode can potentially help novice programmers debug their code, form proper mental models, and explain how their code works to others.

RELATED WORK

Omnicode lies at the intersection of research on program visualization systems and live programming environments.

The most closely-related precursors to Omnicode are program visualization systems [28] such as Jeliot [23], Python Tutor [13], and UUhistle [29]. These tools allow users to write code, run it, and see step-by-step visualizations of stack frames, variables, values, and pointers. Despite their usefulness, these tools all still require the user to run their code and navigate to a specific execution step before inspecting the visualizations. Omnicode takes a more proactive approach by displaying always-on value visualizations live while the user is coding. Also, it is built upon an existing program visualization system (Python Tutor [13]) and embeds those step-level visualizations as a component (Figure 1f).

More distantly related are algorithm visualization (AV) systems [26], which render conceptual visualizations of algorithms rather than concrete run-time values of executing code. Active engagement with AV systems appear to be positively correlated with their effectiveness [17]. In response to these findings, researchers have coupled AV systems with interactive exercises [22] and live environments where users can write pseudocode and see it instantly visualized [16]. Omnicode is inspired by such findings about active engagement and liveness but differs from AV systems because it is designed for working with general-purpose programming languages, not with pseudocode-based algorithm description languages.

Omnicode follows in the long tradition of work in live programming environments [1, 30]. In Tanimoto's classification, Omnicode exhibits Level 3 liveness [30] since its visualizations update in near-real-time whenever the user edits code.

Omnicode innovates upon prior live programming environments by being the first, to our knowledge, to display the full history of run-time values as well as dyadic (two-variable) correlations for a general-purpose imperative programming language. Many live environments have been implemented for declarative or visual languages (e.g., spreadsheets, HyperCard, Forms/3 [6, 32]), where there is no explicit notion of execution history; the current run-time state is always shown and updated live. However, live environments for general-purpose imperative languages with an explicit notion of execution history (e.g., Java [5], JavaScript [20], Python [13], Smalltalk [18], and Lisp [12]) still display only the most recent value for each executed line, expression, or active object – not the entire history of all values. Thus, Omnicode differentiates itself via its always-on live visualizations of all numerical values throughout all executed program steps.

Theseus [21] also experiments with always-on visualizations of run-time state, guided by a similar spirit as Omnicode. However, Theseus visualizes control flow rather than data values since its goal is to inform programmers of which functions in their codebase have executed, how many times, and via what callback functions (which are pervasive in JavaScript web code). Although users of Theseus can select any function call to view its parameter values, it does not provide a comprehensive always-on view of all program values.

Whyline [19] is a powerful debugging tool that logs a comprehensive execution trace of all program values, similar to what Omnicode does. Whyline exposes a query interface for answering "why" and "why not" questions about logged execution state. Although it is engineered to scale far better than Omnicode, its focus is not on providing always-on visualizations of all values, which is infeasible for larger programs, but rather is on acting as a more targeted symbolic debugger.

Omnicode was also influenced by Bret Victor's *Learnable Programming* essay [31], which argues for proactively showing all program run-time state to help novices build mental models. In particular, our visualizations abide by his design principles of "show the data", "show comparisons", and "no hidden state." However, to fully realize his visions from that essay, we may need to redesign programming languages and environments from the ground up rather than simply building visualizations atop existing ones like Omnicode does.

Finally, in terms of UI design, Omnicode's scatterplot matrix is a classic data visualization technique for compactly showing a large set of dual-value correlations [14]. Omnicode also implements classic data interaction techniques [15] such as brushing-and-linking and overview+detail to help users go from skimming over all program values to honing in on those that they want to focus on at a given moment.

THE DESIGN AND IMPLEMENTATION OF OMNICODE

We now revisit the components of Figure 1 with a focus on design motivations and implementation details. We refined the design of Omnicode's interface over several rounds of pilot tests with students in introductory programming courses.

(a) Programming environment: Our current Omnicode prototype is made for Python, but its ideas can potentially transfer well to other general-purpose imperative languages. The instructor first uses the built-in `doctest.py` [2] testing module to write a problem statement, a set of test cases, and skeleton starter code to load into Omnicode. The student launches the Omnicode web app by visiting a URL, selects a test case from the left pane, and starts writing code. They can switch between test cases by clicking on the respective tabs. This kind of self-contained problem-based format is common in university classes, MOOCs, and coding interview prep guides: The student's task is to write a short function to implement a given spec and pass a set of test cases.

(b) Live code execution and run-time trace generation: Omnicode is a live programming environment that always runs the user's code on the currently-selected test case two seconds after they stop typing. Thus, it implements a simple form of *continuous testing* [25]. We found via pilot tests that two seconds was about the right amount of pause time for the experience to still feel "live" but not to seem too jarring by re-running code and updating the UI too frequently.

Omnicode sends the user's code and test case to run on a server. The server uses a modified version of the Python Tutor backend [13] to execute Python code and generate a full run-time value trace. This trace is a list with one element for each executed step: Each element contains all in-scope variables at that step, along with their current types and values. Each

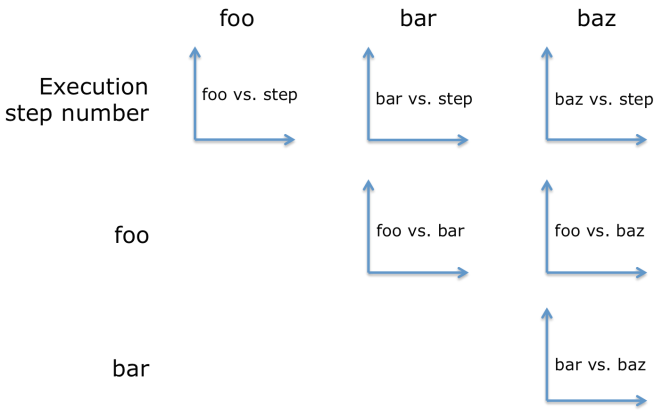

Figure 2. Example scatterplot matrix layout for a program with three numeric-valued variables: foo, bar, and baz. Each variable's value is plotted relative to the execution step number and to all other variables.

variable gets a unique name based on its scope; e.g., local variables are named with their enclosing function name and stack frame index. This trace is encoded as JSON and sent back to the Omnicode web app to render as visualizations. To guard against infinite loops and excessive resource usage, Python Tutor stops execution after 1,000 steps or 3 seconds. Despite its name, the Python Tutor backend also supports executing Java, JavaScript, TypeScript, Ruby, C, and C++, so it is easy to extend Omnicode to those languages in the future.

(c) Scatterplot matrix visualizations: After running the user's code, Omnicode parses the JSON run-time trace from the server and visualizes the values of all numeric-valued variables at all execution steps. To show how each variable changes throughout the entire history of execution, Omnicode generates a set of scatterplots with each individual variable on the y-axis and the execution step number on the x-axis.

This design explicitly surfaces the concept of *variable roles* in programming education [4, 7], which have been shown to be helpful for novices learning to identify deeper structure beneath code execution semantics. Specifically, these scatterplots enable users to visually spot certain single-variable roles such as an integer stepper increasing linearly over time or an accumulator variable's values increasing quadratically. Spotting deviations from expected variable roles could also help novices hone in on logic bugs in their code. Users in our pilot tests found these plots to be the most useful, so we always position them in the first row of the matrix (top of Figure 2).

Omnicode also creates a set of scatterplots plotting the values of each variable against all other variables, which serves to surface dual-variable roles [4, 7] such as one variable always following another in a lock-step pattern. In each of these plots, a data point appears for a given pair of variable values at each execution step. For example, if a variable `foo` is 5 and `bar` is 10 at a particular execution step, then a point appears at (5, 10) in the foo-vs-bar scatterplot. We found in pilots that this kind of visualization was useful for checking expectations about correlations: For instance, the user may expect variables `foo` and `bar` to exhibit a positive linear correlation, but if they see that the foo-vs-bar scatterplot shows zero or negative correlation, then that may indicate a bug.

The scatterplot matrix (SPLOM) is a classic data visualization method that allows us to surface the program's complete run-time value traces in a way that facilitates comparisons throughout time and between dyads. The variable-versus-steps scatterplots were inspired by the "show the data" and "no hidden state" design principles in Bret Victor's *Learnable Programming* essay [31]; the variable-versus-variable plots were inspired by his "show comparisons" principle.

During pilot tests, we also discovered a tension between maintaining a logically coherent layout and users' aversion to scrolling. Our original design (Figure 2) was a fixed-layout matrix: Each column represents one variable, and the first row represents the execution step number while all subsequent rows represent all other variables. The user can quickly glance at the labels for a particular row and column to know what is shown in that scatterplot. However, some users in our pilot tests created too many variables, so their matrix grew too wide to fit on the screen. Based on user feedback, we developed an alternative flowing layout pegged at the monitor's width so that users never need to scroll horizontally; but they must still scroll vertically if there are too many plots. This layout reduces scrolling but makes it harder to tell which scatterplots correspond to which variables. In the current version of Omnicode, users can toggle between the two layouts.

(d) Visualizing custom expressions: Since Omnicode visualizes only numbers by default, one frequent activity we observed in pilot tests was users writing their own custom expressions and assigning them to temporary variables in their code so that they show up in the live visualizations. For instance, users would write code like `y = len(x)` to visualize how the length of a list changes over time and how it is correlated with other variables' values. To accommodate this common use case, Omnicode now allows users to create their own scatterplots by specifying Python expressions to plot on the x and y axes of their plots. This way, they can avoid cluttering their own code with temporary variables.

Custom expressions are sent to the server to evaluate at all execution steps where the indicated variables are in scope. Our current prototype does not guard against side effects, but in the future it could run each expression in a state snapshot so that its side effects do not interfere with subsequent steps.

For convenience, Omnicode automatically derives numeric values from certain non-numeric types such as the size of any collection object and the sum of the values within a list/tuple/set/dictionary (if it contains only numbers). It then proactively adds their value plots to the matrix alongside those for regular program variables (Figure 1d).

(e) Bidirectional brushing-and-linking: The promise of always-on visualizations is that users can quickly get an overview of all run-time values without having to initiate any actions, which opens the possibility of stumbling upon unexpected surprises. When the user does find a point of interest in one of the plots (e.g., a strange-looking dual-value correlation), Omnicode allows them to hone in to see more details.

Figure 3 shows that the user can brush over (i.e., highlight) any 2-D region of any scatterplot, and Omnicode filters all

Figure 3. When the user brushes over a scatterplot, Omnicode filters all other plots to display only the execution steps and corresponding lines of code that fall within the range of that selection.

other plots to show only execution steps that fall within the range of the selection. In addition, Omnicode highlights the executed lines of code in the editor corresponding to those steps. This type of brushing-and-linking allows users to see relationships between more than two variables and also to relate their values back to the relevant lines of code.

Going the other direction, the user can brush over parts of their code to filter the scatterplot matrix. If they select one or more lines of code, that will filter the plots to show only data points resulting from executing those lines. For instance, users can select lines within a loop to see what values were affected by that loop. For example, Figure 1e shows the user selecting line 8 within a loop and the scatterplot matrix hiding irrelevant plots and data points based on that selection.

During pilot tests, we noticed some users trying to select individual variables in their code and expecting that the plots would filter accordingly. In response to this observation, we added support for variable selections in addition to line selections. Omnicode consults location information in the abstract syntax tree (AST) to find which variable the user is (fuzzily) selecting and shows only the scatterplots for that variable.

(f) Detailed visualizations of any data type: The design of Omnicode purposely sacrifices detail for breadth of coverage. However, some pilot test users wanted to zoom in to see more details about execution state at each step and to also inspect non-numerical values. Thus, to complement the always-on scatterplots, we integrated step-level visualizations from Python Tutor. Figure 4 shows that if the user selects a line of code, they see a pop-up pane that shows a detailed visualization of all the steps where that line executed. This is akin to setting a breakpoint in a symbolic debugger, except that runtime state is instantly available in a visual form. These inline visualizations support arbitrary heap object graphs of *all* Python data types (not just numbers), so they offer much more details upon user demand. By incorporating Python Tutor's visualizations, Omnicode allows users to both get a holistic overview using scatterplots and to zoom in on specific steps.

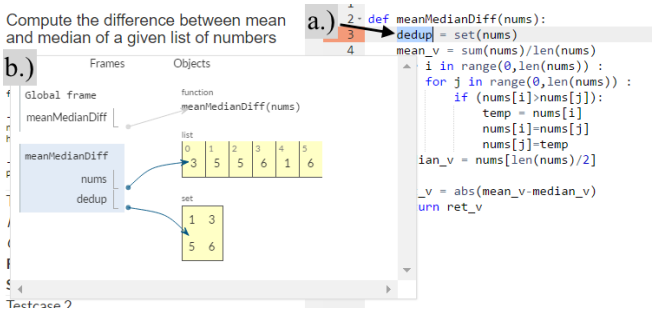

Figure 4. a.) When the user selects a line of code, b.) Omnicode pops up an inline visualization that shows a detailed view of all stack frames, variables, and data structures at each step where that line executed.

EXPLORATORY USER STUDY

Can Omnicode help novices: a) write and debug code, b) form proper mental models, and c) explain their code to others? To take initial steps toward answering these questions, we ran an exploratory first-use study to get novices' first impressions using Omnicode to solve coding problems of the sort seen in introductory programming courses and technical interviews.

Study procedure: We recruited 10 university students (aged 19–26, 2 female) who identified as novice programmers with basic Python familiarity. We brought each subject to our lab for a 75-minute session using Omnicode on a Windows 10 machine with a 34-inch monitor. We first gave a 15-minute tutorial on all of Omnicode's features. We then presented three programming problems involving numerical lists:

- Problem 1: Write a function to find the difference between the minimum and maximum values of a list of numbers.
- Problem 2: Write a function to compute the difference between the mean and median values of a given list of numbers without using Python's built-in sorting functions.
- Problem 3: Write a function to compute the mean of the smallest 25% of values in a given list of numbers.

Each subject had 45 minutes to work on the three problems within Omnicode in whatever order they wanted. We encouraged think-aloud. If they finished early, we had them do an extra task of choosing one of their solutions and verbally explaining its behavior to the experimenter by using Omnicode's visualizations. After the programming session, we spent the final 15 minutes giving them a questionnaire and semi-structured debriefing interview.

Overview of results: Out of our 10 subjects, everyone solved Problem 1 correctly; 6 solved Problem 2; and 9 solved Problem 3. Three completed the extra task of verbally explaining one of their solutions to the experimenter. Table 1 shows the sizes of subjects' final correct solutions for the three problems. Each program had around a dozen lines of code and a half-dozen variables (user-created + automatically derived by Omnicode), which is consistent with the size of basic code written in introductory courses and during whiteboard interviews. Table 2 shows post-study questionnaire results, sorted by mean scores on a 5-point Likert scale from Strongly Disagree (1) to Strongly Agree (5). We developed these questions based on the most salient behaviors and obstacles we

Problem	μ_{lines}	$\mu_{user\_variables}$	$\mu_{derived\_variables}$	μ_{plots}
1	8.30	3.00	2.00	21.10
2	15.33	6.00	2.00	45.67
3	8.67	3.00	2.44	25.11

Table 1. Mean # lines of code, user-created variables, Omnicode-derived variables, and plots in 10 subjects' final code for each problem.

observed during pilot tests. Although these numbers indicate favorable views toward Omnicode's features, it also surfaced concerns about visual overload (which we will detail later). We used questionnaire responses as the basis for our post-study debriefing interview, encouraging subjects to provide qualitative justifications and specific anecdotes to supplement their scores. The first author analyzed interview notes and screen video recordings to group the qualitative user feedback into three main themes, which we now describe in detail:

Omnicode as a Helper for Forming Proper Mental Models

Some of the subjects perceived Omnicode as a helper that runs in the background. In particular, its visualizations helped them to debug code and to reaffirm the correctness of their mental models. S3 said it even helped to better structure their code: *"It motivated me to break complex expressions into smaller ones and store them in variables to view them in the visualization."*

As an example of debugging, when S7 had a bug in his bubble sort algorithm that used a doubly-nested loop over `i` and `j`, he could immediately see in the scatterplots that he had an incorrect mental model of how scoping worked for a `temp` variable declared within the inner loop: *"I just saw that the value of temp actually persists even after the inner for-loop is completed for a given i index; temp was set to 3 when j = 2 and i = 1 then it remained at 3 for j = 0 and i = 2 ... I've always assumed that the value would be set to some other default value when the program finishes executing one inner loop for a given i. I thought after being set to 3, temp would contain some garbage value before getting set with another value later."*

In terms of reaffirming the correctness of mental models, we observed that subjects often glanced at visualizations to clear up any doubts about the results of function calls, type conversion, loop execution, and branch conditions without losing the context and flow of their work. For example, while implementing bubble sort, S3 pointed to a scatterplot and mentioned that *"I can immediately tell that the double loop with i and j variable is hitting every case from the plot with (i, j) values."*

Two subjects proactively used Omnicode to refine their mental models even when there was no obvious bug in their code at the moment. For instance, when S8 was not sure whether numeric division returned an integer or a float, he confirmed it quickly using the visualizations: *"Does sum(lst) / len(lst) return an integer? ... OK it's a whole number in the scatterplot. Now let me multiply 1.0 to the denominator ... Oh, it returned a float this time. Does float() also give the same result? ... Ahh, it does!"* Out of personal curiosity as he was writing bubble sort (unrelated to his main task), S7 spontaneously ran an experiment to see if he could determine whether Python lists were copied by reference or by value: *"Let me assign a list variable lst to newlst. Now I'll append an element to newlst ... oh, the length and sum of both lst and newlst from the execution step a new element was appended changed in the same way ... So it's copy-by-reference for lists!"*

Questionnaire Item	μ	σ
Always-on visualizations helped me verbally explain how my program works.	4.11	1.05
Omnicode's brushing-and-linking helped me filter and focus on most relevant data.	4.11	.60
Always-on visualizations helped me construct correct solutions to programming problems.	3.67	.87
There were too many plots displayed on-screen with uninteresting data points.	3.56	1.13
It was easy to locate the necessary information among all of the plots.	3.22	1.10
Plots were re-rendered too frequently.	2.00	.71

Table 2. Summary of post-study questionnaire responses, averaged over 10 subjects and sorted by mean agreement level on a 5-point Likert scale.

Subjects mentioned an advantage Omnicode had over regular IDEs was that they can instantly test impromptu hypotheses to refine their mental models by simply editing their code and seeing the visualizations. There was no need to write print statements or to wade through screenfuls of text output.

Although we did not perform a rigorous controlled study with print statements or debugger as baselines, anecdotes like the ones presented in this section show some initial promise that being able to see the *entire history of all relevant values* at a quick glance within Omnicode could help novices refine their mental models and hone in on certain kinds of bugs.

Omnicode as an Explanatory and Teaching Aid

Subjects also felt that Omnicode's visualizations were a helpful aid when explaining their program's behavior to others, such as during the extra verbal explanation task in our study. S5 mentioned, *"I think selecting parts of code and seeing how values in the selection change visually over many steps lets us more clearly discuss the program's behavior."*

Subjects liked seeing all scatterplots to provide a high-level overview when starting to explain how their code works to the experimenter. When they needed to explain a part in detail, they could use brushing-and-linking or step-level Python Tutor visualizations. For example, in a loop using an `i` variable, S4 commented: *"By slicing a specific `i` value and brushing over its change, I can see how all the other values change as well; this is a great overview compared to what I would've had to draw on the board if I was teaching."*

In addition to visualizing data values, Omnicode can also show control flow. During S5's study, he used Omnicode's brushing-and-linking to explain an if-statement to the experimenter: *"As you know, the test input was [1,3,2]. Therefore, the if-statement that tests whether the length of the given list is even fails and its body does not execute; you can see from mouse-overing the next step that the code executed was the else-statement body."*

When prompted to explain her code, S9 mentioned parallels from her personal experiences in conducting technical interviews and pair programming sessions: *"Having the visualization was helpful in explaining the behavior of the program in a clearer manner. When I'm interviewing and teaching I often feel like I face a wall due to the lack of common understanding and language between me and the person that I'm communicating with about code. Visualizations can ground these conversations and make them more evidence-oriented."*

These anecdotes show Omnicode's potential as a teaching aid for instructors to visually explain their code demos in class, even if they do not necessarily use it as an IDE to write significant amounts of code. More broadly, a future multi-user extension of Omnicode could be useful in pedagogical settings such as one-on-one tutoring, active learning in the classroom, pair programming, and as a substrate for facilitating deeper discussions during coding-based job interviews.

Visual Overload and Suggestions for Improvement

Unsurprisingly, the main downside of Omnicode that subjects reported was visual overload from seeing too many plots at once. An average of 21 to 45 scatterplots were on-screen by the end of each study (Table 1). Even though they all still fit on a 34-inch monitor with minimal scrolling, subjects felt there were too many plots with uninteresting data and that it was not always easy to locate necessary information (Table 2). The situation would be even worse on a laptop screen. Subjects did not overwhelmingly prefer either the matrix or the flowing layout. The matrix layout made it easier to find a particular plot (Figure 2), but the flowing layout was more compact and did not leave large on-screen gaps.

Besides commenting on the sheer number of plots, some subjects also did not like the large amounts of data points within certain plots. Since all points were shown in the same color, denser plots looked like undifferentiated masses of data.

Subjects made insightful suggestions for coping with visual overload. S1, S2, and S5 suggested merging similar or identical plots together to remove redundancy. S9 suggested color-coding data points to differentiate them based on, say, which code constructs set those values (e.g., a for-loop, if-else). S1 wanted to let the user resize, move, and hide plots. S2 suggested analyzing the run-time traces to automatically detect and highlight "interesting-looking" values and correlations.

However, despite concerns about visual overload, subjects were still satisfied with using brushing-and-linking to hone in on necessary information ($\mu = 4.11$ out of 5 on questionnaire) and were able to solve most of the programming problems.

Study Limitations

The main limitation of our study is that it was exploratory in nature, so we cannot make any rigorous claims about the specific effects of Omnicode's visualizations on comprehension, engagement, or learning. We did not directly compare Omnicode to the status quo of print statements or debuggers, nor to prior live programming environments with different kinds of visualizations. Also, this study involved three small programming problems given in a lab setting; a more ecologically-valid study would be to deploy Omnicode in a university course where students could use it to solve more realistic problems. As such, the findings of this study should be mainly viewed as an *elicitation of design ideas* from our target audience of learners for possible uses in educational contexts. Specifically, our findings raise the possibility of Omnicode's always-on visualizations leading to serendipitous visual discoveries and their uses as communication aids in addition to being debugging tools. These findings could inform follow-up comparative studies or future tool development.

CONCLUSION

We created Omnicode to push on one extreme end of the design space of live programming and program visualization systems by probing the following question: *What if a live programming environment for an imperative language always displays the entire history of all run-time values for all program variables all the time?* We found through our exploratory study that this visually-oriented design – though unconventional – has the potential to be useful as both a debugging and communication aid.

One line of future work that Omnicode has inspired is to explore the role of always-on visualizations in facilitating *self-explanations* [8, 9] amongst novice programmers to help them improve their own understanding of code execution. Researchers in other domains have found that diagrams encourage students to generate more self-explanations [3], which positively affect learning outcomes. But to our knowledge, this has not been explored as much in the context of programming. However, it is important to distinguish between learning about how specific pieces of code operate (which Omnicode and similar tools are well-positioned to do) and learning how programming or programming language semantics work in general (which requires far more scaffolding) [24].

On the flip side, our dogmatic decision to display all numeric values all the time led to some visual overload in our study and would likely prevent Omnicode from scaling to larger programs with, say, dozens of variables and hundreds of plots. Although some subjects wanted us to implement ways to reduce visual overload, others found serendipity *because they could see all historical values at a glance* without the friction of initiating any user actions (e.g., S7 having an "Aha!" moment about variable scoping within nested loops). A more traditional debugger or visualizer interface might have led users to miss out on some of those delightful "Aha!" moments.

One compromise we can pursue in the future is to still keep Omnicode's unique "DISPLAY ALL THE VALUES!" spirit, but to also add automatically-generated suggestions of "interesting" visualizations and code regions for users to focus on. This way, users can still see everything at a glance, but their gaze can be gently directed toward what the system deems as the most interesting. Omnicode 2.0 could determine interestingness based on static code analysis (e.g., variables `i` and `j` always appear close together syntactically), dynamic analysis akin to Daikon [11] (e.g., `i` and `j` always take on strongly-correlated values), or by tracking user interactions within the IDE such as keyboard and mouse locations to determine what the user likely cares about the most at any given moment.

ACKNOWLEDGMENTS

Thanks to the UCSD Design Lab for feedback on early drafts and to the anonymous reviewers for their insightful feedback. This work was supported in part by the National Science Foundation under grant NSF CRII IIS-1660819 (formerly named IIS-1463864).

REFERENCES

1. 2013. A History of Live Programming. http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/. (Jan. 2013).
2. 2017. doctest - Test interactive Python examples. https://docs.python.org/2/library/doctest.html. (2017).
3. Shaaron Ainsworth and Andrea Th Loizou. 2003. The effects of self-explaining when learning with text or diagrams. *Cognitive Science* 27, 4 (2003), 669–681. DOI: http://dx.doi.org/10.1207/s15516709cog2704_5
4. Mordechai Ben-Ari and Jorma Sajaniemi. 2004. Roles of Variables As Seen by CS Educators. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*. ACM, New York, NY, USA, 52–56. DOI: http://dx.doi.org/10.1145/1007996.1008013
5. Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live object exploration: Observing and manipulating behavior and state of Java objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. DOI: http://dx.doi.org/10.1109/ICSM.2015.7332518
6. Margaret M. Burnett, John W. Atwood Jr, and Zachary T. Welch. 1998. Implementing Level 4 Liveness in Declarative Visual Programming Languages. In *Proceedings of the IEEE Symposium on Visual Languages (VL '98)*. IEEE Computer Society, Washington, DC, USA, 126–. http://dl.acm.org/citation.cfm?id=832279.834482
7. Pauli Byckling, Petri Gerdt, and Jorma Sajaniemi. 2005. Roles of Variables in Object-oriented Programming. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 350–355. DOI: http://dx.doi.org/10.1145/1094855.1094972
8. Michelene T.H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. 1989. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science* 13, 2 (1989), 145–182. DOI: http://dx.doi.org/10.1207/s15516709cog1302_1
9. Michelene T.H. Chi, Nicholas De Leeuw, Mei-Hung Chiu, and Christian Lavancher. 1994. Eliciting Self-Explanations Improves Understanding. *Cognitive Science* 18, 3 (1994), 439–477. DOI: http://dx.doi.org/10.1207/s15516709cog1803_3
10. Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. http://www.tandfonline.com/doi/abs/10.1207/s15327809JLS0904_3
11. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.
12. Chris Granger. 2017. Light Table: The next generation code editor. http://lighttable.com/. (July 2017).
13. Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368
14. Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. 2010. A Tour Through the Visualization Zoo. *Commun. ACM* 53, 6 (June 2010), 59–67. DOI: http://dx.doi.org/10.1145/1743546.1743567
15. Jeffrey Heer and Ben Shneiderman. 2012. Interactive Dynamics for Visual Analysis. *Commun. ACM* 55, 4 (April 2012), 45–54. DOI: http://dx.doi.org/10.1145/2133806.2133821
16. Christopher D. Hundhausen and Jonathan L. Brown. 2007. What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing* 18, 1 (2007), 22 – 47. DOI: http://dx.doi.org/10.1016/j.jvlc.2006.03.002
17. Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259 – 290. DOI: http://dx.doi.org/10.1006/jvlc.2002.0237
18. Daniel H. H. Ingalls. 1981. Design Principles Behind Smalltalk. *Byte Magazine* 6, 8 (1981), 286–298.
19. Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 301–310. DOI: http://dx.doi.org/10.1145/1368088.1368130
20. Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *Proceedings of the 2014 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC '14)*. IEEE Computer Society, Washington, DC, USA, 5–8.

21. Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

22. Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppl, and Panu Silvasti. 2004. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. In *Informatics in Education*. 048.

23. Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing Programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, USA, 373–376. DOI: http://dx.doi.org/10.1145/989863.989928

24. Greg Nelson, Benjamin Xie, and Andrew J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 International Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA.

25. David Saff and Michael D. Ernst. 2004. An Experimental Evaluation of Continuous Testing During Development. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 76–85. DOI: http://dx.doi.org/10.1145/1007512.1007523

26. Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. 2010. Algorithm Visualization: The State of the Field. *Trans. Comput. Educ.* 10, 3, Article 9 (Aug. 2010), 22 pages. DOI: http://dx.doi.org/10.1145/1821996.1821997

27. Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. DOI: http://dx.doi.org/10.1145/2483710.2483713

28. Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Trans. Comput. Educ.* 13, 4, Article 15 (Nov. 2013), 64 pages. DOI: http://dx.doi.org/10.1145/2490822

29. Juha Sorva and Teemu Sirkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 49–54. DOI: http://dx.doi.org/10.1145/1930464.1930471

30. Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617346

31. Bret Victor. 2012. Learnable Programming: Designing a programming system for understanding programs. http://worrydream.com/LearnableProgramming/. (Sept. 2012).

32. E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-manipulation Programming Systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 258–265. DOI: http://dx.doi.org/10.1145/258549.258721