The Essence of Program Semantics Visualizers: A Three-Axis Model

Abstract

A program semantics visualizer (PSV) helps illuminate a language's semantics by explaining the runtime execution of programs. PSVs are often used in introductory programming (CS1) courses to help introduce a notional machine, an abstraction of the computer that executes the language. But what information should PSVs present to fully explain such notional machines?

In this paper we propose a three-axis model to assess the design of PSVs that visualize execution traces. PSVs should help users by clearly answering three questions: What is the machine's configuration at each execution step? Why did an execution step take place? How did an execution step change the machine's configuration? We demonstrate our model's utility for assessing PSVs by explaining why, in actual classroom use, instructors have resorted to manually extending Python Tutor's visualizations.

2012 ACM Subject Classification

Keywords and phrases Program Semantics Visualizer, CS1, Notional Machine, Abstract Machine, Term Rewriting System, Python Tutor

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

Students struggle to form accurate mental models of program execution. For example, in a study of an introductory Java programming course, only 17% had an accurate mental model of object reference assignment [20]. The authors identified five distinct incorrect mental models among the other 83%.

Developing an accurate mental model is essential, since even understanding simple programs demands mastery of several crucial semantic concepts. For example, consider the Python program in Listing 1. Understanding this code involves (at least) four aspects of Python's semantics: expression evaluation, variable lookup, function entry, and loops.

These and other semantic concepts compose a notional machine, "an idealized abstraction of computer hardware and other aspects of the runtime environment of programs" [27]. Program semantics visualizers (PSVs), like Python Tutor [16], visualize traces of program execution to help explain notional machines by example [10] (Figure 1).

Getting a PSV's design "right" is vital for helping users develop accurate mental models. But how can we assess whether a PSV explains everything a student needs to comprehend the necessary semantic concepts?

Listing 1 Factorial in Python. Understanding this five-line function requires mastering (at least) four semantic concepts: expression evaluation, variable lookup, function entry, and loops.

```python
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i + 1)
    return product

print(fact(6))```

© Author. Please provide a copyright holder; licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:15

OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Motivated by theories of visual expressiveness (Subsection 2.1) and leveraging abstract machine formalizations (Subsection 2.2), we contribute a three-axis model for assessing PSV design. Specifically, we identify three key questions (Section 3) a PSV should clearly answer:

1. **What** is the machine’s configuration at each step of execution? (Subsection 3.1)
2. **Why** did an execution step take place? (Subsection 3.2)
3. **How** did an execution step change the machine’s configuration? (Subsection 3.3)

We demonstrate the utility of this model by accounting for instructors’ manual extensions to Python Tutor’s visualizations. Of the four semantic concepts necessary to understand Listing 1, our model accounts for extensions to explain expression evaluation, variable lookup, and function entry. For extensions concerning loops, we found that our model does not elegantly explain all the extensions we observed in practice. We conclude that our model is sufficient for visualizations of single execution steps. Future work will need to build on the foundation established in this paper with additional axes to account for explanations that visualize multiple steps simultaneously. We speculate how such extensions may be developed (Section 5) and urge others to develop tools that clearly answer PSV users’ “What, Why, How” questions automatically (Section 6).

In the spirit of PLATEAU, our work repurposes insights from programming languages (PL) to advance visualization design and ultimately help democratize programming. Specifically, our three-axis model leverages formal semantics to address the needs of three user groups: (i) it suggests how learners may develop accurate mental models of programs, (ii) it provides guidelines for visualization authors who customize PSVs, and (iii) it provides design criteria for PSV tool builders.

## 2 Formalizing the Visualization Task

### 2.1 Visual Expressiveness

We frame our model in terms of visual expressiveness [21]. Mackinlay phrases visualization as a problem of converting a set of facts to a visual representation that is more easily digestible by a user. From this perspective, he is able to provide a formalization for part of what makes some visualizations better than others:

**Visual Expressiveness:** “A set of facts is *expressible* in a language if it contains a sentence that (1) encodes all the facts in the set, (2) encodes only the facts in the set.”
To convey a set of facts to a user, a PSV’s visual grammar must be able to encode it. PSVs aim to communicate program semantics to users in a way that is easier to digest than formal semantics expressed as mathematical rules. To assess the visual expressiveness of a PSV, then, we must provide a formal definition of the set of facts in a program’s semantics.

### 2.2 Abstract Machines

To the best of our knowledge, Berry [5] was the first to explore the role of formal semantics in PSVs. That early work used big-step and small-step operational semantics, which require additional animation steps that do not correspond to operations in the original machine rules. Sirkiä [26] introduced Jsvee, a language-agnostic DSL for creating PSVs. Though Jsvee gives tool authors freedom to design animations and explanations, it is not opinionated about the information these visualizations contain. In particular, the primitives do not answer “Why” questions about execution.

Pollock et al. [25], on the other hand, suggest that abstract machines can play a central role in formally reasoning about PSVs. Abstract machines present machine models close to existing semantics visualizations and provide a well-defined notion of time step. Using Haskell as a meta-language, below we present a formal definition of an abstract machine based on Abstract Evaluation Systems [17].

Intuitively, an abstract machine is a combination of (1) a set of possible machine configurations, including both initial configurations corresponding to a program beginning execution on a given input and also final configurations corresponding to program results, and (2) a transition function that explains how configurations evolve over time as the machine executes. For brevity, we elide the details of initial and final configurations below.

In practice, abstract machine transition functions exhibit common structure. We follow Hannan and Miller [17] and represent transition functions as a collection of rewrite rules:

```haskell
class AbstractMachine op where
    rules :: [Rewrite op]

data Rewrite a = ...
```

---

1 While the code used for our formalism is executable and precisely describes the model, it is intended only for reasoning and illustration, not for use in real-world implementations.
Here \( \text{op} \) is the type of function operators over machine configurations (\( \text{Config} \), detailed below), and \( \text{rules} \) is the collection of rewrite rules that specify the transition function.

A rewrite \( l \rightarrow r \) consists of a left-hand pattern, \( l \), and a right-hand pattern, \( r \). For example, the rule \( x + x \rightarrow 2 \times x \) has \( l \) as \( x + x \) and \( r \) as \( 2 \times x \). To apply this rule to \( 1 + 1 \), we first match the \( l \) against the term, yielding the substitution \( x \mapsto 1 \). Then we apply the substitution to \( r \), yielding the new term \( 2 \times 1 \). In our case, rewrites will always match on the entire program configuration, not on any nested subterms.

To keep our formalism simple, we assume a common restriction on rewrite rules: orthogonality [17]. A collection of rewrite rules is orthogonal if it satisfies two properties. First, the rules must be left-linear: variables can only appear once in the left-hand pattern. The example rewrite rule above is not left-linear. However, a rewrite rule such as \( x + y \rightarrow y + x \) is left-linear. Second, the rewrites must be non-overlapping: for any term, only a single rewrite rule in the collection will match.

The simplicity of rewrite systems provides generality [17], neatly representing many common abstract machines including CEK, SECD, Krivine, and CAS-based semantics.²

### 3 The Three-Axis Model

Beginning with our basic abstract machine formalization, we simultaneously motivate each of our model’s three axes and refine our machine description.

#### 3.1 Trace: Answering What? Questions

Ben-Bassat Levy et al. [19] propose completeness as a design goal for PSVs: “Completeness means that every feature of the program must be visualized, for example, a value such as a constant may not appear from nowhere.” We reformulate this definition of completeness as a question about notional machines that PSVs must help users answer:

**What** is the machine’s configuration at each step of execution?

A PSV can answer **What?** questions if it can present all intermediate configurations to the user in detail. We formally model this as **trace**: the transitive closure of configurations reached by the transition function’s rewrites starting from an initial configuration and ending in a final configuration. The trace from initial configuration \( c_0 \) to final configuration \( c_n \) is the sequence \( c_0, c_1, \ldots, c_n \) such that the transition function rewrites \( c_i \rightarrow c_{i+1} \). Using Haskell as our meta-language, we formally specify **trace** as:

```haskell
import Data.List (find); import Data.Maybe (isJust); import Control.Arrow ((>>>))

data Config a = Node a [Config a]

-- run 'step' until it fails, collecting intermediate results
trace :: AbstractMachine op => Config op -> [Config op]
trace c = c : iterateUntilNothing step c
  where iterateUntilNothing :: (a -> Maybe a) -> a -> [a]
        iterateUntilNothing f = unfoldr (f >>> fmap (\x -> (x, x)))
```

² Language features such as machine arithmetic and capture-avoiding substitution require special treatment in this model; however, neither of these posed issues in our analysis. We believe this model can be extended to support those features without fundamentally changing the axes.
-- Find the rewrite that successfully runs on 'c'. Return 'Nothing' if none do.
step :: AbstractMachine op => Config op -> Maybe (Config op)
step c = firstJust (map (runRewrite c) rules)
  where firstJust = join . find isJust

-- Run the rewrite on the config, returning a new config if it succeeds.
runRewrite :: Eq a => Config a -> Rewrite a -> Maybe (Config a)
runRewrite = ...

The list of rules of the abstract machine specifies its transition function. step tries to
apply these rules to advance the machine configuration one step. If it successfully applies a
rewrite, it returns Just of the new configuration. If it doesn’t, step returns Nothing. step,
via rules, specifies the semantics of our abstract machine. Thus to understand an abstract
machine’s semantics, one must understand the step function. But functions are abstract
and thus difficult to visualize, especially statically. PSVs make the choice to demonstrate
step by visualizing execution traces (Figure 2). That is, they initialize the abstract machine,
call step repeatedly until it stops, then present the intermediate states to the user. The
trace function runs step until it returns Nothing and produces all the intermediate machine
configurations, yielding a trace which can be visualized to answer What? questions.

3.2 Pattern Match: Answering Why? Questions

Nelson et al. [23] argue that students must also understand “the causal relationship between
syntax and machine behavior”. That is, why does syntax cause a particular evolution in
the machine? Rather than providing an explicit definition, the authors define causality
using an implicit virtual machine model with syntax, bytecode instructions, and machine
configurations. We reformulate this definition of causality as a question about notional
machines that PSVs must help users answer:

Why did an execution step take place?

In our general abstract machine formalism, each execution step is driven by a rewrite from
the transition function’s rules. Thus, to answer Why? questions, a PSV must help students
understand how step decides which rule to run. Again, we generalize this in our formalism
using Haskell as a meta-language to detail matching:

data Rewrite a = Rewrite {lhs :: Pattern a, rhs :: Pattern a}
data Pattern a = Cnstr a [Pattern a] | Var String
type Subst a = [(String, Config a)]

match :: Eq a => Pattern a -> Config a -> Maybe (Subst a)
apply :: Subst a -> Pattern a -> Config a

runRewrite :: Eq a => Config a -> Rewrite a -> Maybe (Config a)
runRewrite n (Rewrite {lhs=left, rhs=right}) = do
  subst <- match left n
  apply subst right

match :: Eq a => Pattern a -> Config a -> Maybe (Subst a)
match (Var x) n = Just [(x, n)]
The match function contains the logic for determining which rule fires, since it only returns a mapping if the match succeeds. To determine whether or not a particular rewrite rule fires, match compares the left-hand pattern of that rule against the current machine configuration. If the pattern is a constructor that matches the configuration’s constructor, we visit its children and repeat, otherwise the match fails. If the pattern is a variable, we add the corresponding piece of the configuration to the substitution map.

To discuss the cause of a particular rule firing, we use the notion of counter-factual or “actual” causality [22]. Roughly, we define causality to mean that A causes B iff A precedes B and if A didn’t happen then B didn’t happen. Though this definition poses philosophical issues in general settings, in our restricted case, we can apply it in a fairly straightforward way. We wish to “blame” some pieces of the machine configuration for a rule firing. If we look at the Cnstr case of match, we see that changing the contents of a Node will directly change whether or not that rewrite rule fires. Thus the pieces of config that match Cnstr nodes cause a particular rewrite rule to fire.

For example, imagine we have the rewrite rules $x + y \rightarrow y + x$ and $x \times y \rightarrow y \times x$ and we evolve the configuration $1 + 2$ to $2 + 1$. + causes this transition, because the pattern $x + y$ that matched the configuration contains a single constructor, +. Changing it to $\times$ would result in the other rule firing. However, changing $1 + 2$ to $1 + 3$ does not change which rule fires, because it is not changing part of the configuration that is matched by a Cnstr.

Notice this definition relies on the non-overlapping assumption to ensure that changing data matched by the variable components of a pattern will never cause a different rule to match.

Summing up, a PSV can answer Why? questions if it can explain how match decided to execute a given rewrite rule. It can do this by identifying what pieces of configuration were matched by constructors in the pattern.

### 3.3 Pattern Application: Answering How? Questions

Just as students must understand the causes of a rule firing, they must also understand its effects. Ben-Bassat Levy et al. [19] suggest the principle continuity as a design goal for PSVs: “Continuity means that the animation must make the relations between actions in the program explicit. For example, Jeliot 2000 [(the authors’ PSV)] shows how the values of the subexpressions of an expression contribute to its value. This means that the visual objects that represent subexpressions must remain visible until all of them have been evaluated; then these objects are animated to form the expression.” We reformulate this definition of continuity as a question about notional machines that PSVs must help users answer:

**How** did an execution step change the machine’s configuration?

A PSV can answer How? questions if it can explain how the abstract machine uses information from the previous configuration to construct the next configuration. PSVs can do this by identifying what pieces of the previous configuration this step’s rewrite retains, where those pieces go, and which pieces of the configuration the rewrite simply drops. Again, we generalize this in our formalism using Haskell as a meta-language to detail rewriting:
import Data.Maybe (fromJust)

apply :: Subst a -> Pattern a -> Config a
apply s (Cnstr c ps) = Node c (map (apply s) ps)
apply s (Var x) = fromJust (lookup x s)

Just as \texttt{match} encodes information about why a rule executed, \texttt{apply} encodes information about how data is constructed in the new state. \texttt{apply} uses the substitution map and the right-hand side pattern of a rule to build a new program configuration. If the right-hand side pattern is a constructor, we use that data to build a node and visit its children. If the pattern is a variable, we look it up in the substitution, which copies data from the previous configuration. \texttt{apply} fails if a variable does not exist in the substitution map. We assume that all rewrite rules will be “well-formed” in the sense that this lookup will never fail (all variables in the right-hand side pattern must also exist in the left-hand side pattern). This ensures that only \texttt{match} makes decisions about which rule succeeds.

\texttt{apply} constructs data in two ways. In the \texttt{Cnstr} branch it creates data based on the contents of the \texttt{rhs} pattern. In the \texttt{Var} branch it copies data from the previous state. These two actions encode the effects of a rewrite. Data can also be destroyed in two ways. If a variable is matched in the \texttt{lhs} pattern, but not used in the \texttt{rhs} pattern, that data disappears. Similarly, concrete pieces matched in the \texttt{lhs} pattern are not in the substitution map and thus completely “forgotten” during the rewrite.

Thus PSVs can answer \textbf{How?} questions by illustrating how \texttt{apply} introduces, moves, or eliminates information from the previous configuration to construct the next configuration.

\section{Case Study: An Assessment of Python Tutor in the Wild}

Motivated by formal abstract machines, causality, and visual expressiveness, the previous sections detailed our three-axis model of the information PSVs should encode to help students develop accurate mental models of notional machines. To demonstrate the utility of our model, we use it to explain instructors’ manual extensions of Python Tutor visualizations.

\subsection{Method}

To assess a PSV in the wild, we used our three-axis model to analyze uses of Python Tutor in introductory programming (CS1) courses. We focused on Python Tutor because it has attracted millions of users during its first 10 years \cite{1} and because several university courses and textbooks rely on it. Crucially, though many instructors use Python Tutor as a PSV, it was originally designed to be a visual debugger. This means that rather than using Python’s semantics as its ground truth, Python Tutor aims to visualize the information encoded in PDB \cite{16}, a line-level debugger. Because of this discrepancy in purpose, we used the notion of operational misfits \cite{6} to help determine core elements of Python visualizations in the classroom. Instructors bridge the gaps between Python Tutor’s PDB-based visualizations and Python’s underlying semantics with visual annotations, auxiliary explanations, and completely custom diagrams. We call these operational misfits additions.

\textbf{Corpus.} We first assembled a corpus of instructor additions. We examined all 81 CS1 courses at the 40 most prominent CS departments in the U.S. \cite{15} and identified three that used Python Tutor. Within these courses, we identified slides with explanations of the four
semantic concepts we identified in Listing 1: expression evaluation, variable lookup, function entry, and loops. For each semantic concept, we collected visually distinct additions, totalling 18 unique slides. Figure 3 shows examples of additions for each of these concepts.

**Analysis.** We attempted to explain the information in each addition using our three axes. We analyzed both visual and textual information.

- **What:** Does the addition include machine configuration data or execution steps that are present in Python’s semantics, but not in Python Tutor?
- **Why:** Does the addition explain why a rule executed?
- **How:** Does the addition show how data moves from one configuration to the next?

Additions that could not be fully explained with our axes were marked Other.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>CS61A</td>
<td>UC Berkeley</td>
<td>Expr Eval</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S2</td>
<td>CSE160</td>
<td>Univ of Washington</td>
<td>Expr Eval</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S3</td>
<td>CSE160</td>
<td>Univ of Washington</td>
<td>Var Lookup</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S4</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Var Lookup</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>CS61A</td>
<td>UC Berkeley</td>
<td>Var Lookup</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S6</td>
<td>CS61A</td>
<td>UC Berkeley</td>
<td>Func Entry</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Func Entry</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S8</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Func Entry</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Func Entry</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>S10</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Func Entry</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>S11</td>
<td>CS61A</td>
<td>UC Berkeley</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S12</td>
<td>CSE160</td>
<td>Univ of Washington</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S13</td>
<td>CSE160</td>
<td>Univ of Washington</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S14</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S15</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S16</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S17</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S18</td>
<td>CSIC0040</td>
<td>Brown Univ</td>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1 We analyzed 18 slides from three courses that used Python Tutor visualizations. We labeled each addition with the axes used by the instructor to improve Python Tutor’s output, and marked additions with unexplained components as “Other.” Axes used are strongly correlated within semantic concepts, which suggests our model identifies the additional information required to understand these concepts rather than changes in visual presentation. Additions to loops visualizations answered global questions across multiple execution steps rather than the local, single-step questions in our model. We propose extensions to our model that could incorporate this information in Section 5.

### 4.2 Results

Table 1 summarizes our collection of instructors’ explanations. For the first three concepts we analyzed, we found that additions for the same concept usually employed similar axes. Expression evaluation explanations added What information about how expressions become values. Variable lookup explanations added Why information to explain how Python chose what scopes to look for variables. Function entry explanations added How information to show how arguments move from a function call to a function body. We believe this
While Statements

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Figure 3 Example figures from our corpus for each of the four semantic concepts. In each example, instructors add additional annotations to customize Python Tutor visualizations. They use many different techniques to add information including (from upper left) custom data structures, text, arrows, and strikethroughs. Despite this diversity, additions for the same semantic concept generally employ the same axis of our model. Expr Eval S1 [9], Var Lookup S4 [14], Func Entry S6 [8], Loops S11 [9]

Our three axes formalize the information encoded in a single execution step. We discuss extensions of our model to multiple execution steps in Section 5.

Implications. These results suggest that our three-axis model is useful for identifying the information required to understand single steps of program execution. Our analysis of loop additions suggests instructors use multi-step explanations to provide intuition for more complex semantic concepts.

3 S1, S2, S8, S11, S14, S15, S16, S17, S18
Future Work: Towards Higher-Level Semantic Explanations

Abstract Interpretation. We hypothesize multi-step visualizations such as those employed by loops could be formalized using abstract interpretations that glue together multiple individual steps. For example, Lerner [18] connects loop summary visualizations to collecting semantics, which, for each location in the source program, “collects” the configurations the machine executes through while at that location. This prompts further questions about the role abstraction plays in PSVs. We believe our model, with its connections to PL and its granular treatment of individual execution steps, is well-suited to these kinds of extensions.

Programs As Term Rewrite Systems. We note that our definition of abstract machine, while specific enough to extract concrete aspects of visual expressiveness, is actually general enough to apply to systems beyond low-level program semantics. In fact, a program in a lazy functional language, like Haskell, may be interpreted as a rewrite system [28]. In the words of the authors of one such system: “A Clean program basically consists of a number of graph rewrite rules (function definitions) which specify how a given graph (the initial expression) has to be rewritten” [24]. This connection suggests that explanations and visualizations of programs more generally may also benefit from answering our What?, Why?, and How? questions.

Conclusion

In this paper we have proposed a three-axis model—What? Why? How?—for assessing the design of single-step PSVs. We believe these axes will help ensure students build robust mental models of notional machines by leaving fewer student inferences about program semantics to chance.

But what about creating PSVs in the first place? Because we have based these axes on an executable formalism, we suspect one could generate answers to the three questions directly from abstract machine definitions or any other term rewriting system. We imagine a world in which, rather than poking around in the dark with a printf flashlight to understand programs, formal systems can explain themselves.

References

1 Visualize code execution learn python, java, c, c++, javascript, and ruby. URL: http://pythontutor.com/.
7 John DeNero. Environments. URL: https://inst.eecs.berkeley.edu/~cs61a/sp20/assets/slides/05-Environments_full.pdf.
Anonymous Authors

8 John DeNero. Names, assignment, and user-defined functions. URL: https://inst.eecs.berkeley.edu/~cs61a/sp20/assets/slides/02-Names_full.pdf.
9 John DeNero. Print and none. URL: https://inst.eecs.berkeley.edu/~cs61a/sp20/assets/slides/03-Control_full.pdf.

11 Jason Gaudette. Lecture 01 introduction to python. URL: http://cs.brown.edu/courses/csci0040/lectures/lec01.pdf.
12 Jason Gaudette. Lesson 03 iteration in python. URL: http://cs.brown.edu/courses/csci0040/lectures/lec03.pdf.
15 Philip Guo. Python is now the most popular introductory teaching language at top us universities. BLOG@ CACM, July, 47, 2014.
The Essence of Program Semantics Visualizers


**A Appendix: Corpus of Instructor Additions**

**Discussion Question 1 Solution**

```
func min(...)
func max(...)

f(2, g(h(1, 5), 3))
g(h(1, 5), 3)
```

**Another evaluation example**

```
(72 – 32) / (9.0 * 5)
(40) / (9.0 * 5)
40 / (9.0 * 5)
40 / (45.0)
40 / 45.0
.888
```

**Figure 4** Expression Evaluation Examples S1 [9], S2 [4]
How to look up a variable

Idea: find the nearest variable of the given name

1. Check whether the variable is defined in the local scope
2. ... check any intermediate scopes ... (none in CSE 160) ...
3. Check whether the variable is defined in the global scope
   If a local and a global variable have the same name, the global variable is inaccessible ("shadowed")
   This is confusing; try to avoid such shadowing

```python
x = 22
stored = 100
def lookup():
    x = 42
    return stored + x
val = lookup()
x = 5
stored = 200
val = lookup()  # What happens if we define stored after lookup?
```

• When the call to mystery2 is about to return:

Visualizing How Functions Work
pythontutor.com/visualize.html

- Python looks for a variable in the current frame first, so the local x will be used instead of the global x when returning x + 1.

Local Names are not Visible to Other (Non-Nested) Functions

- An environment is a sequence of frames.
- The environment created by calling a top-level function (no def within def)
  consists of one local frame, followed by the global frame.

Figure 5 Variable Lookup Examples S3 [3], S4 [14], S5 [7]
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):
1. Add a local frame, forming a new environment
2. Bind the function’s formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Visualizing How Functions Work

pythontutor.com/visualize.html

• At the start of the call to mystery2:

  mystery2(3, 2)

  Gets its own frame containing the variables that belong to it.

  mystery2’s x isn’t shown yet because we haven’t assigned anything to it.

What is the output of this code?

```
def calculate(a, b):
    a = y
    b = x + 1
    return a * b

print(calculate(3, 2))
```

A. 5
B. 9
C. 4
D. 3
E. 8

The values in the function call are assigned to the parameters.
In this case, it’s as if we had written:
```
x = 3
y = 2
```

Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)
```

```
def f(x):
    return 11*g(x) + g(x//2)
```

```
def g(x):
    return -1 * x
```

Figure 6 Function Entry Examples S6 [8], S7 [14], S8 [11], S9 [14], S10 [14]
**Figure 7** Loops Examples S11 [9], S12 [2], S13 [2], S14 [13], S15 [12], S16 [12], S17 [12], S18 [13]