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

Josh Pollock
MIT CSAIL, USA
https://joshmpollock.com
jopo@mit.edu

Grace Oh
International School, Bellevue, WA, USA
https://github.com/Gracesoh
grace27sw.oh@gmail.com

Eunice Jun
University of Washington, USA
https://homes.cs.washington.edu/~emjun
emjun@cs.washington.edu

Philip J. Guo
UC San Diego, USA
https://pg.ucsd.edu
pg@ucsd.edu

Zachary Tatlock
University of Washington, USA
https://ztatlock.net
ztatlock@cs.washington.edu

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 PSV output. In particular, we study instructors' additions to visualizations generated by Python Tutor, the most popular PSV.

2012 ACM Subject Classification Human-centered computing → Visualization theory, concepts and paradigms

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

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1836813. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

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

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:19

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.

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

print(fact(6))
```

of object reference assignment [19].

Yet 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.

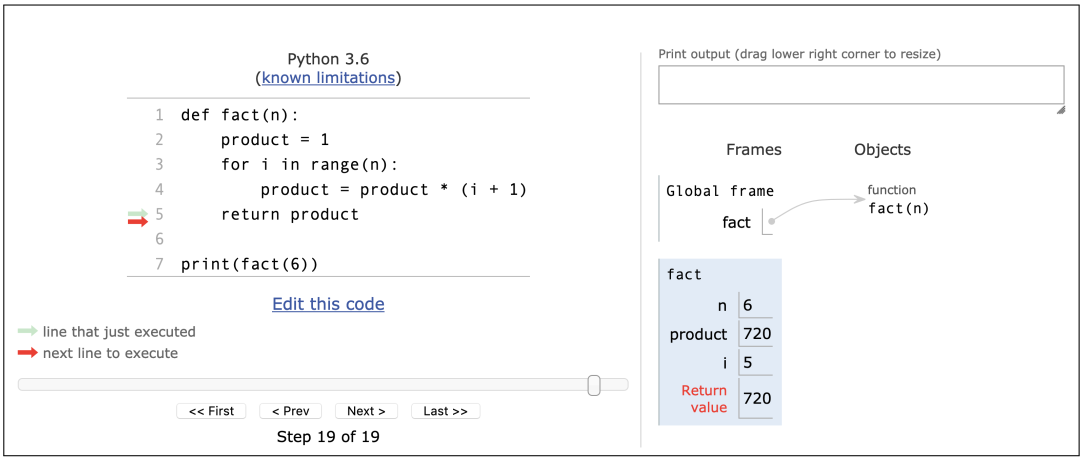


Figure 1 Python Tutor visualization of the factorial program in Listing 1.

These and other semantic concepts compose a *notional machine*, "an idealized abstraction of computer hardware and other aspects of the runtime environment of programs" [25]. *Program semantics visualizers (PSVs)*, like Python Tutor [15], visualize traces of program execution to help explain notional machines by example [9] (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?

Leveraging abstract machine formalizations (section 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. More specifically we study extensions from the first few lectures of three CS1 courses, whose content comprises the four semantic concepts necessary to understand Listing 1. Our model accounts for extensions that explain expression evaluation, variable lookup, and function entry. For extensions concerning loops, we found that our model does not elegantly account for instructors' explanations we observed in practice. We conclude that our model is sufficient for visualizations of single executions 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*.

More precisely to create an expressive PSV, we must be able to construct a mapping from a machine model to a visual domain. Since we are doing this in a language-agnostic way, we will not use a specific machine model example, but rather grab some language-agnostic definition. Programming Languages offers many possibilities, but the one that suits our needs best is the *abstract machine*.

2 Abstract Machines

By definition a PSV visualizes program semantics. Thus to understand the information a PSV should explain, one must study the information intrinsic to the semantics themselves. In this section we explain our choice of abstract machines as the formal basis for our study of program semantics.

To the best of our knowledge, Berry [5] was the first to explore the role of formal semantics in PSVs. That early work relied on big-step and small-step operational semantics. While this proved to be a useful formalism, operational semantics are generally not conducive to visualizations, since their inference rules often rely on the evaluation of subterms. The ordering of subterm evaluation is implicit, requiring additional animation steps that do not correspond to operations in the original machine rules. Sirkiä [24] introduced Jsvee, a language-agnostic DSL for creating PSVs. Jsvee provides roughly 50 high-level semantic building blocks, called "operations," which are shared among many programming languages. These primitives are useful in practice for quickly developing PSVs. Our approach is complementary, establishing a simpler, more general model of PSV design that we believe could inform the design of Jsvee's semantic blocks.

In contrast to the approaches above, Pollock et al. [23] suggest that abstract machines could 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. Starting here and continuing through the next section, we will incrementally present a formal definition of an abstract machine based on *Abstract Evaluation Systems* [16]. 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 relation* that explains how configurations evolve over time as the machine executes. For brevity, we depart from *Abstract Evaluation Systems* by eliding the details of initial and final configurations. Together these properties comprise a *labeled transition system*:

▶ **Definition 1.** *A* labeled transition system *is a triple* $\langle C, \Lambda, \rightarrow \rangle$ *where* C *is a set of configurations,* Λ *is a set of labels, and* $\rightarrow \subseteq \Lambda \times C \times C$ *is a labeled transition relation. One often writes the label above the arrow:* $\xrightarrow{L} \subseteq \{L\} \times C \times C$.

We can think of the labels as different rules that combine to fully describe the machine's execution.

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. [18] 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 information as a *trace*: the transitive closure of configurations reached by the transition relation starting from an initial configuration and ending in a final configuration. For simplicity, we assume execution is deterministic. Most PSVs operate on specific linear traces, so this is a reasonable restriction. 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 abstract machine relates c_i to c_{i+1} by some rule L_i. That is, $c_i \xrightarrow{L_i} c_{i+1}$.

This formalism suggests **What?** questions are not sufficient, since they say nothing about the L_i. In the next two subsections, we will explore design principles and questions that address *relations* between states.

3.2 Pattern Match: Answering *Why?* Questions

Nelson et al. [21] argue that students must also understand "the causal relationship between syntax and machine behavior". That is, why does a syntax fragment 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?

To address this question, we must refine our model. Configurations alone do not contain the information necessary, rather, we must look to the machine's transition relation for answers. In our general abstract machine formalism, each execution step is driven by a label. Definition 1 merely characterizes the relation as a collection of opaque rules. Presenting the relevant rule to the user could help provide an answer to this question, but an abstract rule is no better than the formal semantics themselves. We need to refine our definition.

3.2.1 Term Rewriting

In practice, abstract machine rules exhibit common structure. By refining our definition to reflect this structure, we can provide a more detailed answer to **Why?**. We follow Hannan and Miller [16] and refine our labeled transition relation definition to a *term rewriting system*:

▶ **Definition 2.** *A* term *is a* `Node` *consisting of a name, na, and list of subterms, ns. We denote the term by* $\texttt{Node}(na, ns)$*.*

▶ **Definition 3.** *A* pattern *is either a variable name,* $\texttt{Var}(x)$*, or a constructor consisting of a name, ca, and subpatterns, ps. We denote a construct pattern by* $\texttt{Cnstr}(ca, ps)$*.*

▶ **Definition 4.** *A* rewrite rule *is a pair* $\langle LHS, RHS\rangle$ *of patterns. To rewrite a configuration* c *into a new configuration* c' *using a rewrite* r*, we* match *the* LHS *against* c *to get a substitution map from variables in the pattern to values, then* apply *that substitution to* RHS *to build* c'*. We stylize a rewrite as* $LHS \rightsquigarrow RHS$*.*

For example, the rule $x + x \rightsquigarrow 2 \times x$ has LHS as $x + x$ and RHS as $2 \times x$. To apply this rule to $1 + 1$, we first match the LHS against the term, yielding the substitution $x \mapsto 1$. Then we apply the substitution to RHS, yielding the new term 2×1. In our case, rewrites will always match on the *entire* program configuration, not on any nested subterms.

▶ **Definition 5.** *A* term rewriting system *is a pair* $\langle C, R\rangle$ *where* C *is a set of configurations and* R *is a set of rewrite rules. If a rewrite rule* R_i *matches a configuration* c_i *and produces a configuration* c_{i+1}*, we write* $c_i \xrightarrow{R_i} c_{i+1}$*.*

Though rewrite systems refine transition systems, they are still general [16], neatly representing many common abstract machines including CEK, SECD, Krivine, and CAS-based semantics.[1]

3.2.2 Using Patterns to Answer Why Questions

To answer **Why?** questions, a PSV must help students understand why one pattern matched and others did not. We describe the match phase (introduced in Definition 4) of rewriting in pseudocode, to study how this decision is made.

To keep our causal analysis simple, we assume a common restriction on rewrite rules: orthogonality [16]. 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, but a rule such as $x + y \to y + x$ is. Second, the rewrites must be *non-overlapping*: for any term, only a single rewrite rule in the collection will match. Left-linearity makes the pattern matching algorithm straightforward and could be relaxed. Non-overlapping rules are easier to reason about causally as we will see below and also ensure deterministic execution.

[1] 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.

input : pattern and configuration
output : A substitution if the pattern matches.

```
match(p, n):
switch (p, n) do
    case (Var(x), _) do
        return Some([(x, n)])
    end
    case (Cnstr(ca, ps), Node(na, ns))
     do
        if ca == na then
            return match(ps, ns)
        else
            return None
        end
    end
end
```

input : patterns and configurations
output : A substitution composed of the match on each pair only if they all succeed.

```
match(ps, ns):
switch (ps, ns) do
    case ([], []) do
        return Some([])
    end
    case ([p, ...ps], [n, ...ns]) do
        switch (match(p, n), match(ps, ns))
         do
            case (Some(s), Some(ss)) do
                return Some(s ++ ss)
            end
            case _ do
                return None
            end
        end
    end
end
```

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 [20]. 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 $x + y \to y + x$ and $x \times y \to y \times x$ 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. [18] 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. As with **Why?** questions, we formalize this principle by analyzing rewrite rule pseudocode:

```
input  : substitution and pattern
output : A new term created by plugging the substitutions into the pattern.
apply(s, p):
switch p do
    case Var(x) do
        return s.lookup(x)
    end
    case Cnstr(c, ps) do
        return Node(c, ps.map(apply(s)))
    end
end
```

Just as `match` encodes information about *why* a rule executed, `apply` encodes information about *how* data is constructed in the new state. `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 variable, we look it up in the substitution, which copies data from the previous configuration. If it is a constructor, we use that data to build a node and visit its children.

`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 `match` makes decisions about which rule succeeds.

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

Thus PSVs can answer **How?** questions by illustrating how `apply` introduces, moves, or eliminates information from the previous configuration to construct the next configuration.

4 Case Study: An Assessment of Python Tutor in the Wild

Motivated by formal abstract machines and existing informal PSV design principles, 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.

4.1 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 [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 [15], a line-level debugger. 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 augmentations *instructor additions*, and they highlight information gaps we hypothesized our model could explain.

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. [14] 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 and that comprise the majority of early content in CS1 courses: expression evaluation, variable lookup, function entry, and loops. For each semantic concept, we collected visually distinct additions, totalling 18 unique slides, which are listed by concept in Appendix A.

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**.

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 (Appendix A.1). Variable lookup explanations added **Why** information to explain how Python chose what scopes to look for variables (Appendix A.2). Function entry explanations added **How** information to show how arguments move from a function call to a function body (Appendix A.3). We believe this correlation between semantic concepts (which are further linked to semantic rules of the abstract machine) and axes indicates our model reasons about semantic content rather than surface-level choices about how information is presented.

Other. While we could explain most of the additions for the first three concepts, some parts of those explanations and all of the loop explanations did not use our three axes

ID	Course	School	Topic	What?	Why?	How?	Other
S1	CS61A	UC Berkeley	Expr Eval	✔	-	-	✔
S2	CSE160	Univ. of Washington	Expr Eval	✔	-	-	✔
S3	CSE160	Univ. of Washington	Var Lookup	-	✔	-	-
S4	CSCI0040	Brown University	Var Lookup	-	✔	-	-
S5	CS61A	UC Berkeley	Var Lookup	-	✔	-	-
S6	CS61A	UC Berkeley	Func Entry	-	-	✔	-
S7	CSCI0040	Brown University	Func Entry	-	-	-	✔
S8	CSCI0040	Brown University	Func Entry	-	-	✔	✔
S9	CSCI0040	Brown University	Func Entry	-	-	✔	-
S10	CSCI0040	Brown University	Func Entry	✔	-	✔	-
S11	CS61A	UC Berkeley	Loops	-	-	-	✔
S12	CSE160	Univ. of Washington	Loops	-	-	-	✔
S13	CSE160	Univ. of Washington	Loops	-	-	-	✔
S14	CSCI0040	Brown University	Loops	-	-	-	✔
S15	CSCI0040	Brown University	Loops	-	-	-	✔
S16	CSCI0040	Brown University	Loops	-	-	-	✔
S17	CSCI0040	Brown University	Loops	-	-	-	✔
S18	CSCI0040	Brown University	Loops	-	-	-	✔

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 loop 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.

of information. Nine of the slides[2] (six from loops) presented information from multiple execution steps at once. 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.

S12 and S13 rewrote more complex code into simpler code students already had a mental model of. For example, S13 unrolled while loops to straight-line code that students already knew how to execute. This deviates from the underlying semantics of Python; however, this visualization could be modeled by a different, but equivalent, abstract machine. Finally, S7 explained why information *did not* appear rather than why it did; and S17 represented the program as a flowchart to explain loops.

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.

[2] S1, S2, S8, S11, S14, S15, S16, S17, S18

5 Future Work: Towards Higher-Level Semantic Explanations

Abstract Interpretation. We hypothesize that abstract interpretation could formalize multi-step visualizations, such as those used to describe loops. For example, Lerner [17] 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 Rewriting Systems. We note that our definition of abstract machine, while specific enough to identify three distinct kinds of information, 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 [26]. 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" [22]. This connection suggests that explanations and visualizations of programs more generally may also benefit from answering our **What?**, **Why?**, and **How?** questions.

Implementation Challenges. Our work contributes a conceptual model to guide PSV design, and our pseudocode formalization suggests that one could generate answers to the three questions *directly* from abstract machine definitions or any other term rewriting system. However, to put our axes into practice, PSV researchers must solve additional challenges. PSVs inspired by our model may require new debuggers that track not only state information at each execution step, but also how information flows between states. To visualize this information, designers must also develop easy ways to render programs' diverse collections of data.

6 Conclusion

In this paper we proposed a three-axis model — *What? Why? How?* — for critiquing the information presented in single-step PSVs. Based on our evaluation of instructor annotations, we expect these axes can improve PSVs so that students can build more robust mental models of notional machines.

Using term rewriting systems as a formal basis for our model, we have also suggested these axes are applicable in more general contexts than teaching low-level program semantics. 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/`.
2 Ruth Anderson. Control flow: Loops. URL: `https://courses.cs.washington.edu/courses/cse160/20wi/lectures/02-loops-20wi.pdf`.
3 Ruth Anderson. Functions and abstraction. URL: `https://courses.cs.washington.edu/courses/cse160/20wi/lectures/04-functions-20wi.pdf`.
4 Ruth Anderson. Introduction to python and programming. URL: `https://courses.cs.washington.edu/courses/cse160/20wi/lectures/01-intro-python-20wi.pdf`.
5 Dave Berry. *Generating program animators from programming language semantics.* PhD thesis, University of Edinburgh, 1990.

6 John DeNero. Environments. URL: `https://inst.eecs.berkeley.edu/~cs61a/sp20/assets/slides/05-Environments_full.pdf`.

7 John DeNero. Names, assignment, and user-defined functions. URL: `https://inst.eecs.berkeley.edu/~cs61a/sp20/assets/slides/02-Names_full.pdf`.

8 John DeNero. Print and none. URL: `https://inst.eecs.berkeley.edu/~cs61a/sp20/assets/slides/03-Control_full.pdf`.

9 Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. Engage against the machine: Rise of the notional machines as effective pedagogical devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 159–165, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3341525.3387404`.

10 Jason Gaudette. Lecture 01 introduction to python. URL: `http://cs.brown.edu/courses/csci0040/lectures/lec01.pdf`.

11 Jason Gaudette. Lesson 03 iteration in python. URL: `http://cs.brown.edu/courses/csci0040/lectures/lec03.pdf`.

12 Jason Gaudette. Lesson 04 more iteration, nested loops. URL: `http://cs.brown.edu/courses/csci0040/lectures/lec04.pdf`.

13 Jason Gaudette. Making decisions: Conditional executions. URL: `http://cs.brown.edu/courses/csci0040/lectures/lec02.pdf`.

14 Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, 47, 2014.

15 Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 579–584, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2445196.2445368`.

16 John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

17 Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3313831.3376494`.

18 Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A Uronen. The jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, jan 2003. URL: `https://doi.org/10.1016%2Fs0360-1315%2802%2900076-3`, `doi:10.1016/s0360-1315(02)00076-3`.

19 Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, 39(1):499–503, March 2007. `doi:10.1145/1227504.1227481`.

20 Peter Menzies and Helen Beebee. Counterfactual theories of causation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2020 edition, 2020.

21 Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 2–11, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3105726.3106178`.

22 Marinus J Plasmeijer. Clean: a programming environment based on term graph rewriting. *Electronic Notes in Theoretical Computer Science*, 2:215–221, 1995.

23 Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. Theia: Automatically generating correct program state visualizations. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, SPLASH-E 2019, page 46–56, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3358711.3361625`.

24 Teemu Sirkiä. Jsvee & kelmu: Creating and tailoring program animations for computing education. *Journal of Software: Evolution and Process*, 30(2):e1924, 2018.

25 Juha Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Educ.*, 13(2), July 2013. URL: https://doi-org.offcampus.lib.washington.edu/10.1145/2483710.2483713, doi:10.1145/2483710.2483713.

26 Y. Toyama, S. Smetsers, M. V. Eekelen, and R. Plasmeijer. The functional strategy and transitive term rewriting systems. 1993.

A Appendix: Corpus of Instructor Additions

A.1 Expression Evaluation

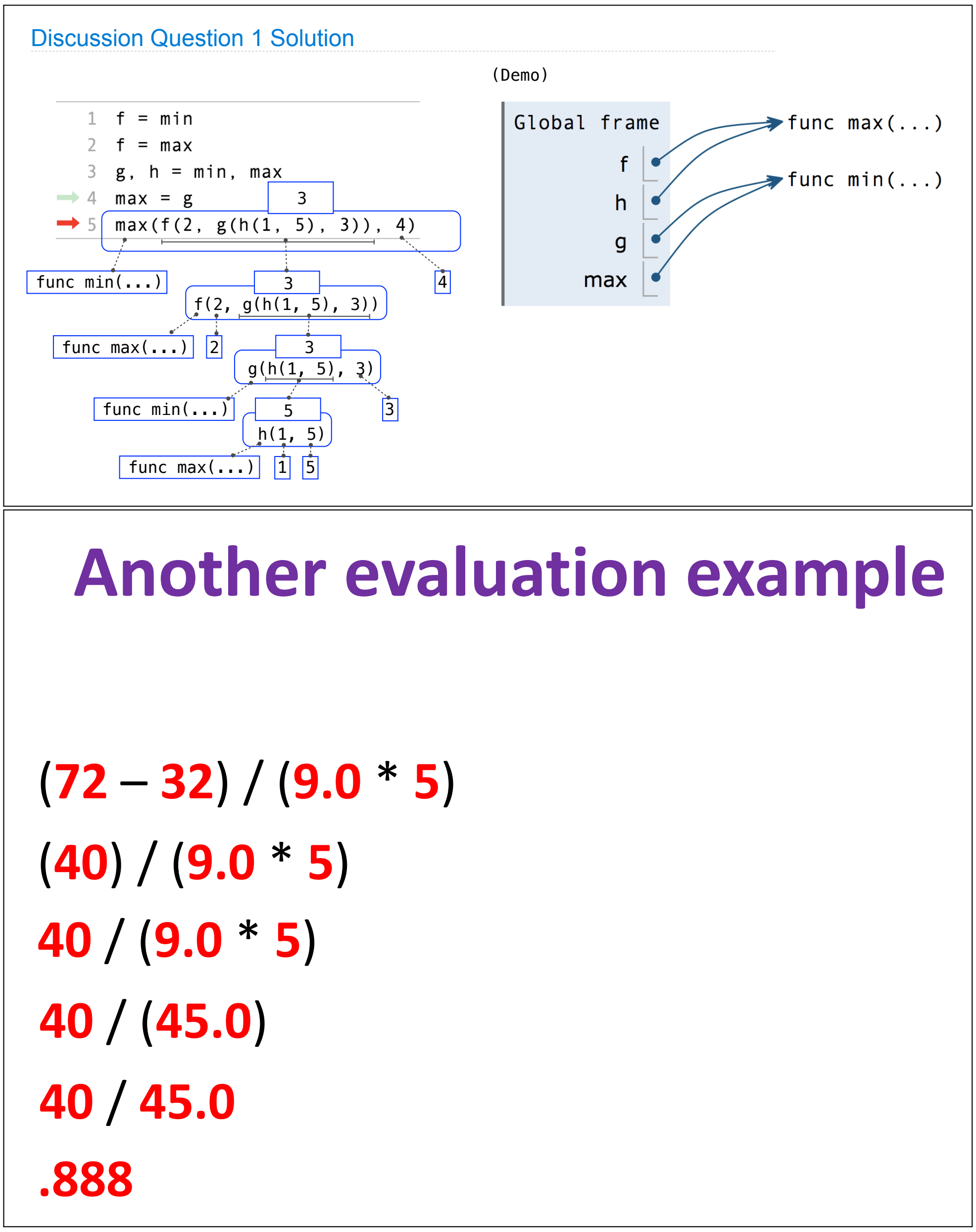


Figure 2 Expression Evaluation Examples S1 [8], S2 [4]

A.2 Variable Lookup

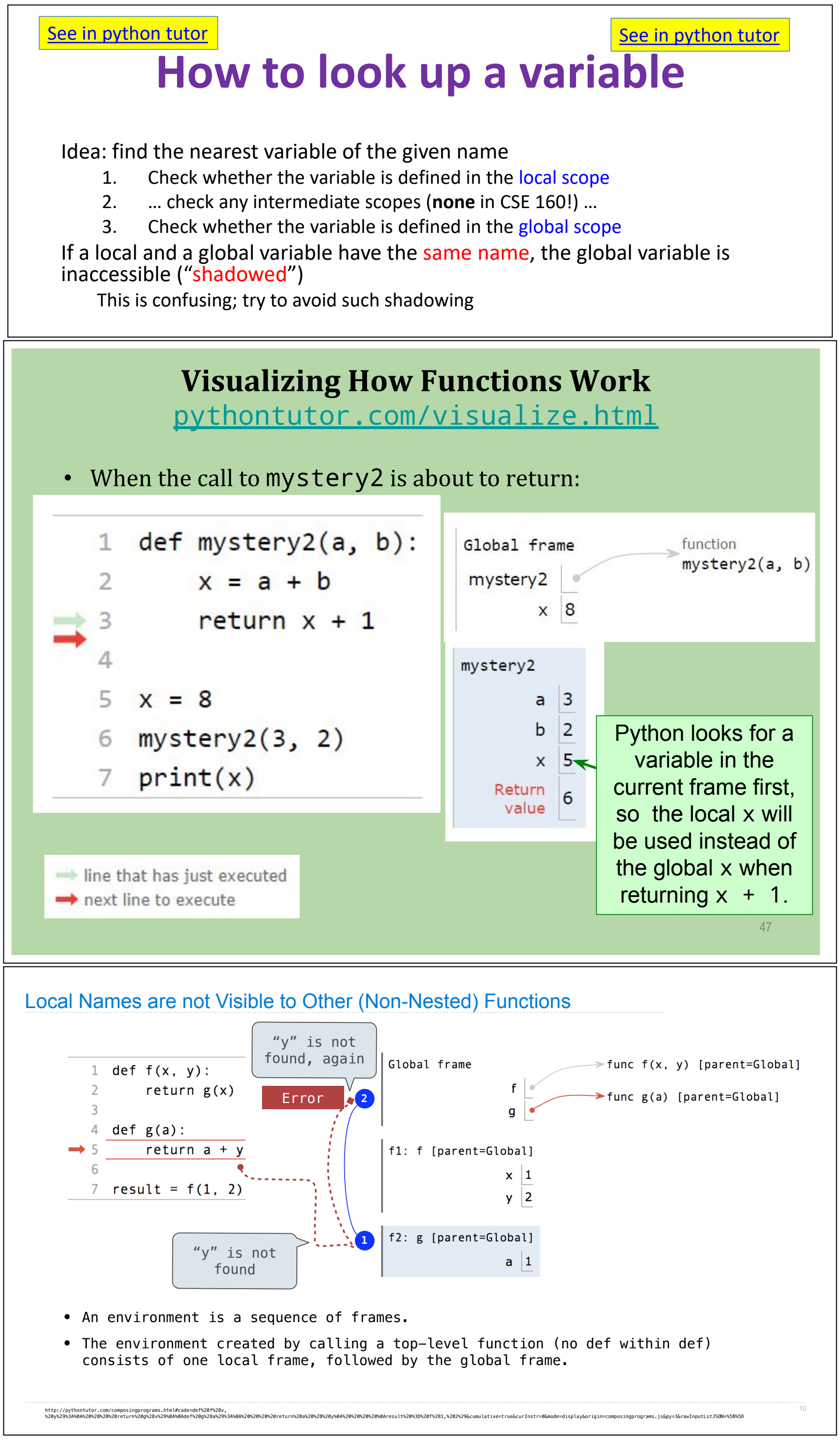


Figure 3 Variable Lookup Examples S3 [3], S4 [13], S5 [6]

A.3 Function Entry

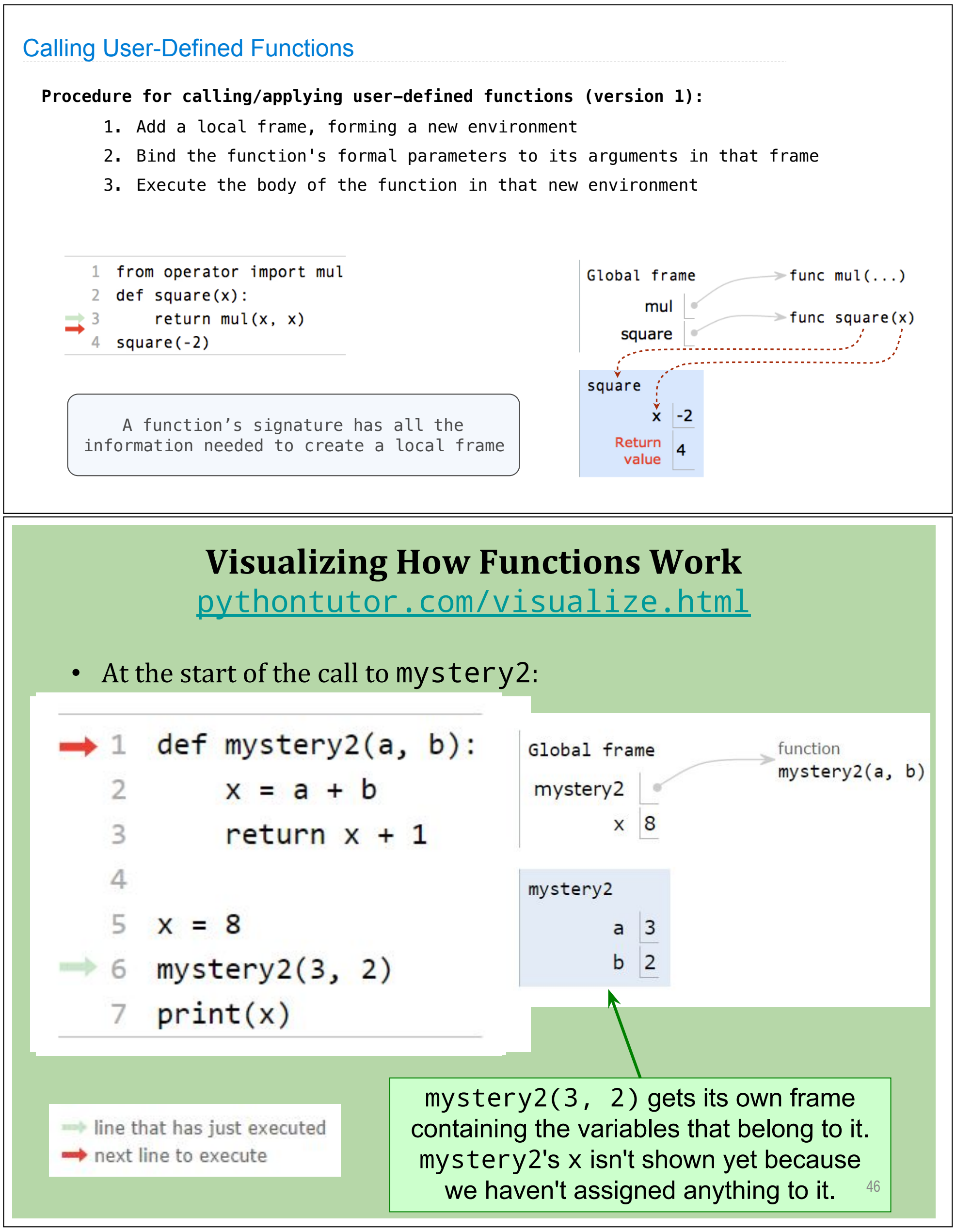
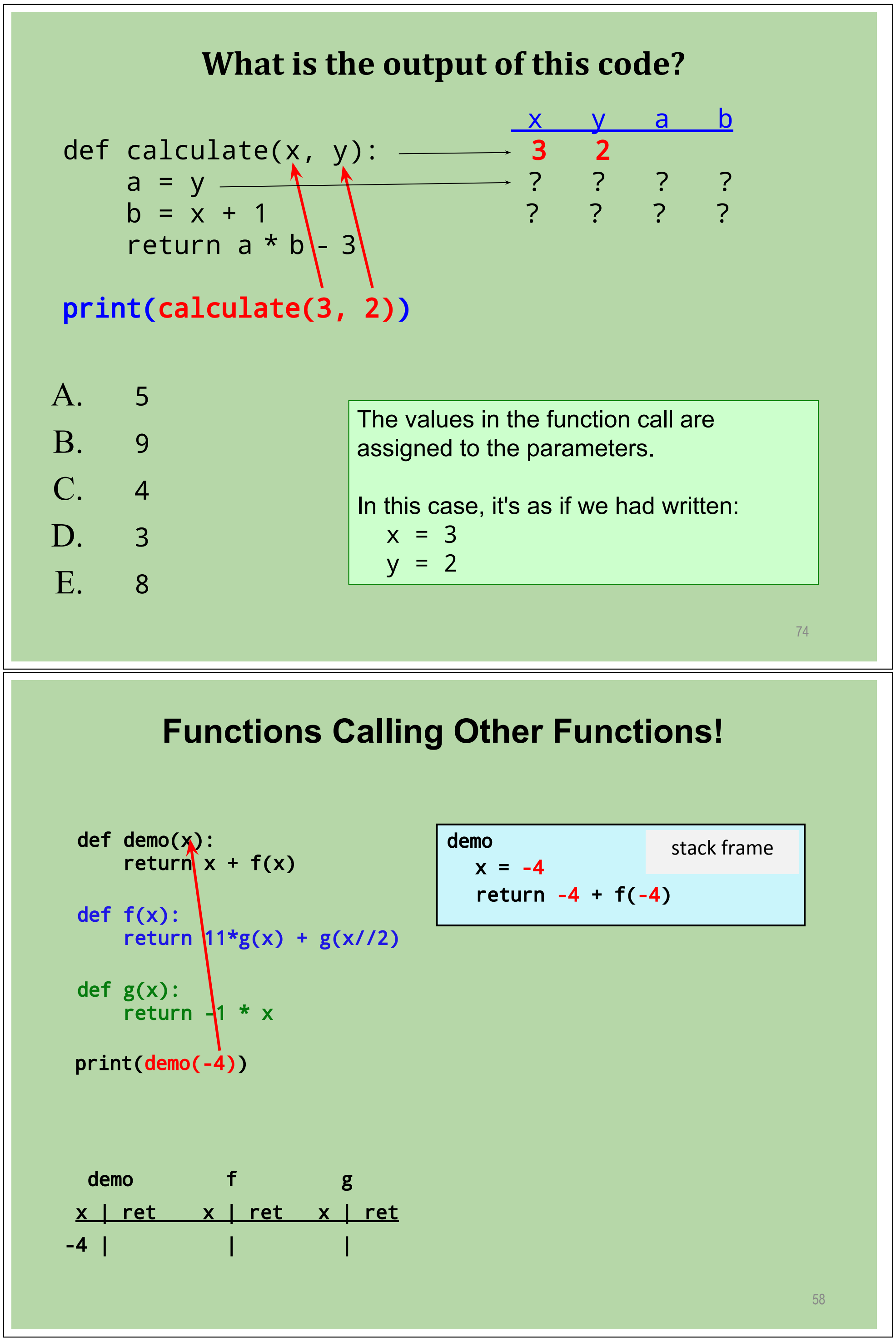


Figure 4 Function Entry Examples S6 [7], S7 [13],



■ **Figure 5** Function Entry Examples S8 [10], S9 [13],

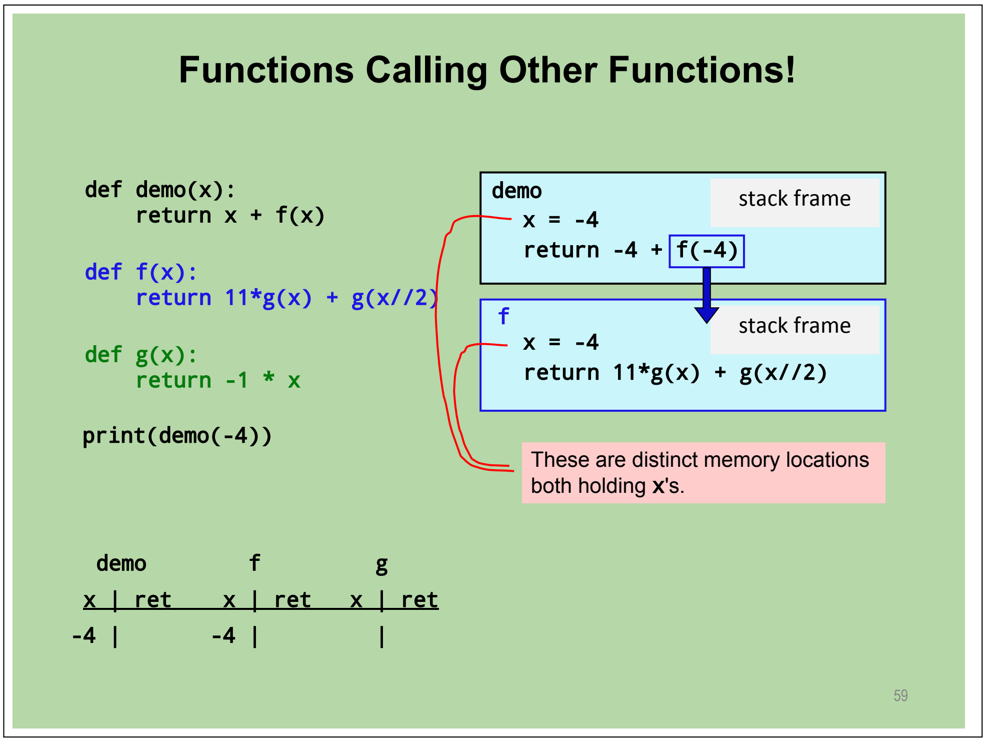


Figure 6 Function Entry Examples S10 [13]

A.4 Loops

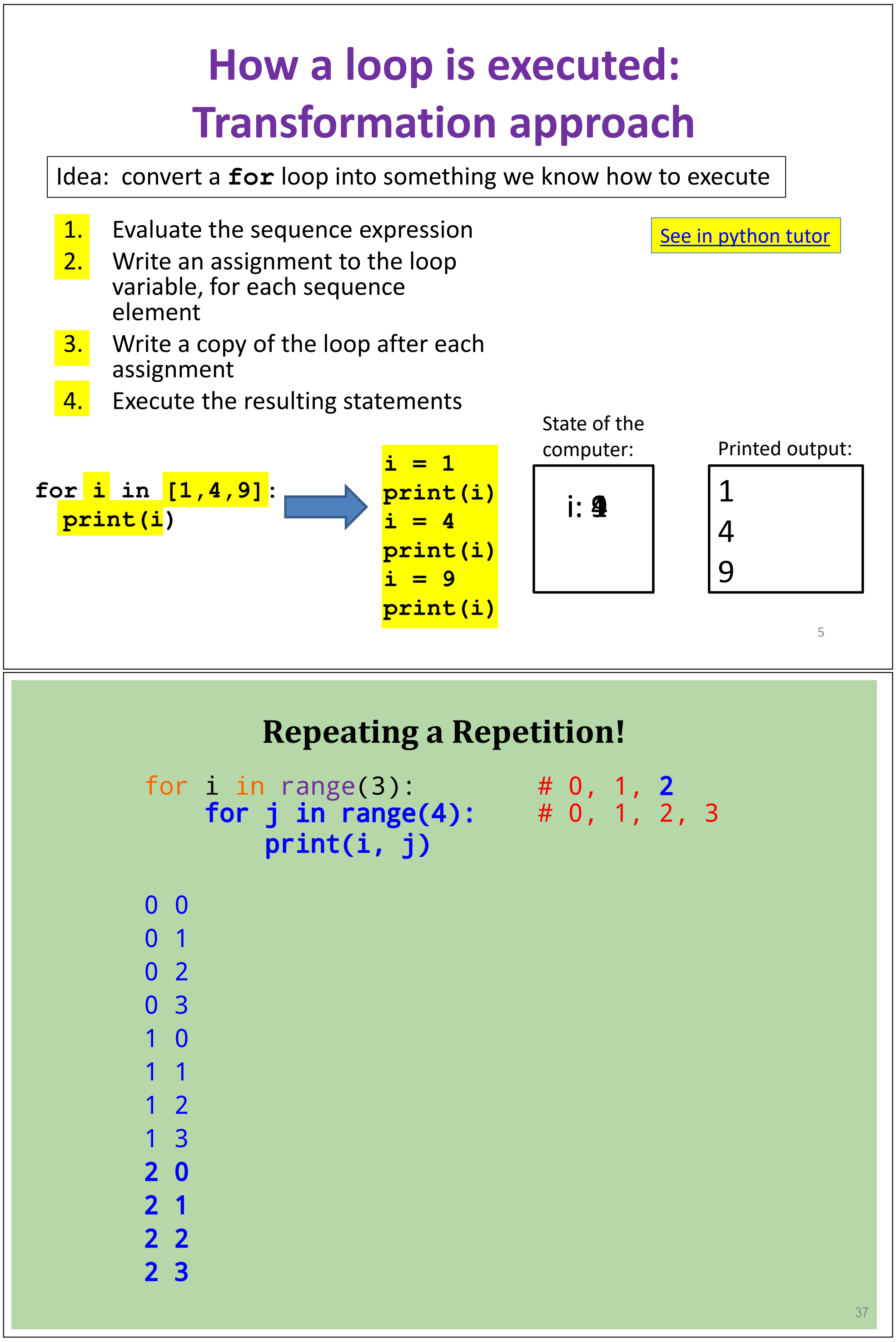
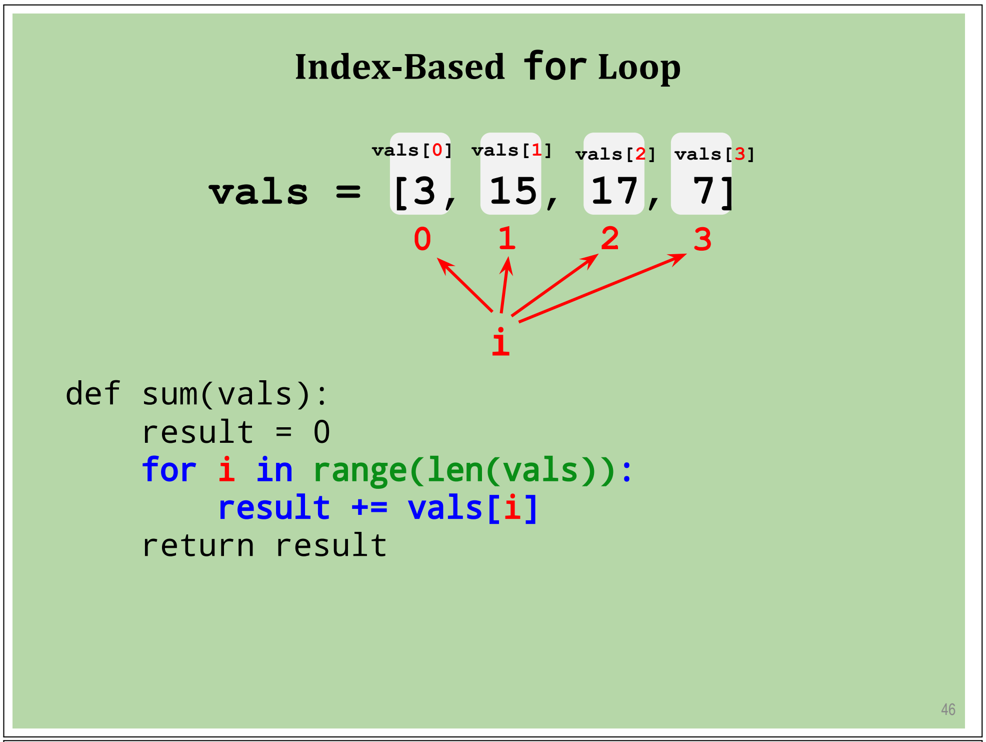


Figure 7 Loops Examples S13 [2], S14 [12],



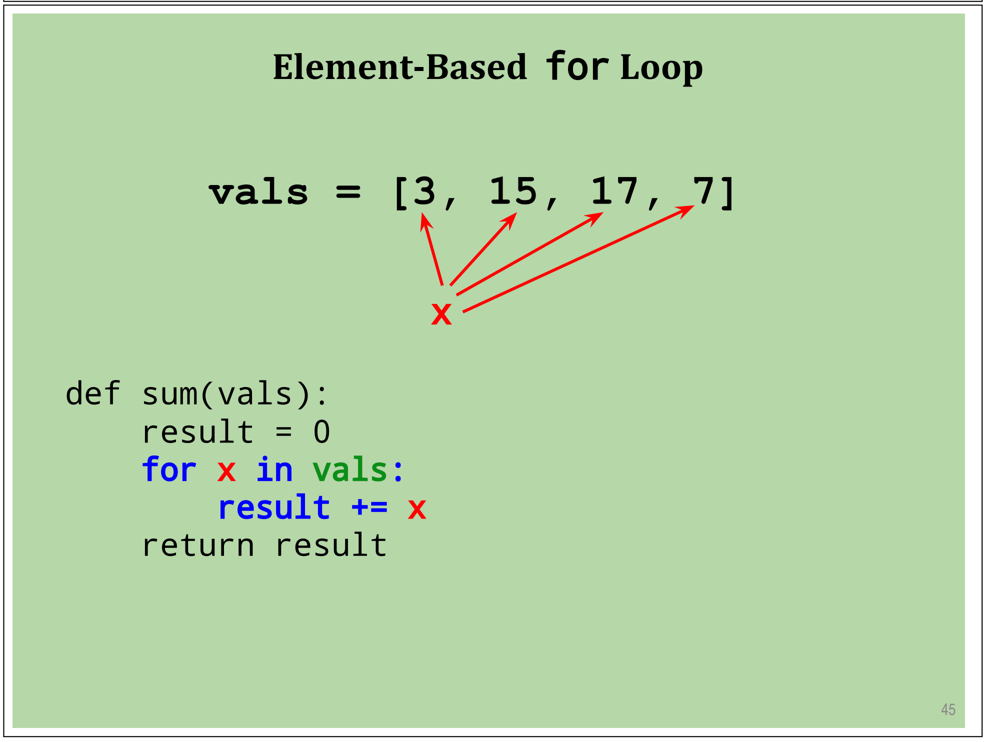


Figure 8 Loops Examples S15 [11], S16 [11],

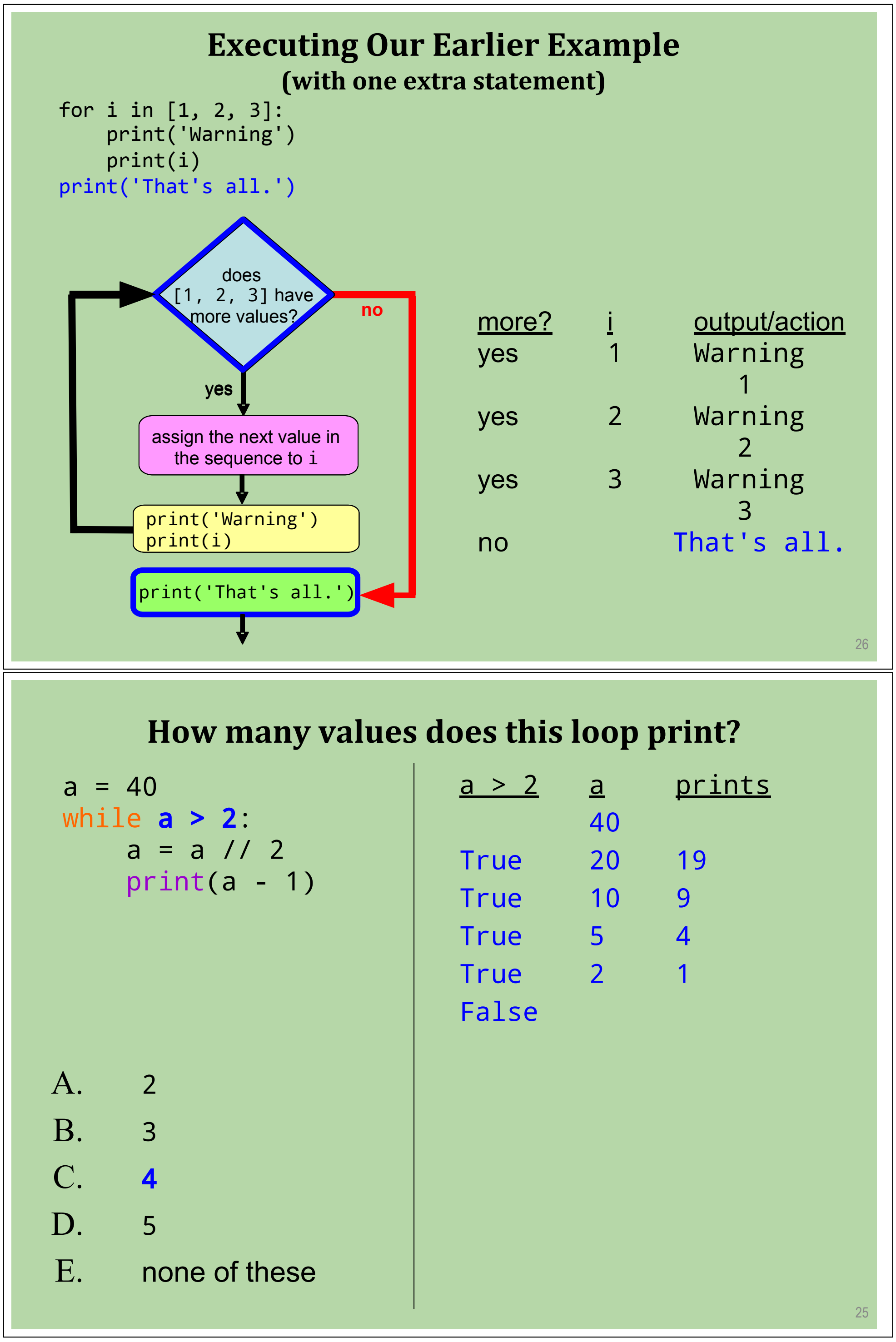


Figure 9 Loops Examples S17 [11], S18 [12]