Curricular and community resources: Supporting Scripting for All*

Lilly Lee, Hallie Seay, Zachary Dodds
Harvey Mudd College
Claremont, CA 91711
{lglee, hseay, dodds}@g.hmc.edu

Abstract

This work envisions resources that help all of an institution's undergraduates build a foundation of computational authorship. Here we present materials evolved from many years of experience requiring Intro-to-Computing (Comp1) of all first-semester students. We hope to prompt and join other institutions looking for ways to engage as much of their undergraduate cohort as possible in computing.

1 Context-and-Community Tools: Developing shared computational models

Every direction we look, our era offers opportunities for computing to contribute. Whether through intensive calculations or insight-producing summaries, computing offers accessible, repeatable, executable interaction-models. From their patterns and dynamics, deeper insights can emerge and fundamental relationships can be discovered or reinforced.

The creation of conceptual models is at the heart of computing. Exploring such models is the realm of introductory computing; our institution requires a "Comp1" course of all first-semester students. This universality has prompted us to develop and customize curricular tools that emphasize students' "shared computing-experience" and promote student-support of each year's new, incoming cohort. This work highlights new directions along this path.

*Copyright ©2021 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

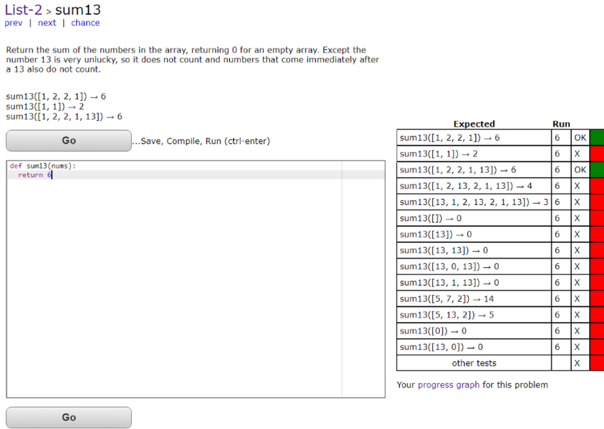

Figure 1: CodingBat interface. It is a clean, simple website with a code editor and test cases with which you can check your code.

2 Coding with Wally: A "multiple-path" CodingBat

For many years our students have used CodingBat, a venerable – and wonderful – set of small-function exercises by Nick Parlante and co-authors at Stanford University[3]. Students are prompted to compose a function matching desired input-output behavior. Once run, students see the test cases they have passed with others, perhaps, failed. Sometimes some of the test cases are hidden.

CodingBat's approach is enormously valuable! It's with good reason that it has become so widespread: to solve a problem, one internalizes the specification, perhaps tries a few examples, understands it, and expresses a solution. Codingbat's interface supports this workflow well.

3 The opportunity: Additional "approach headings" to computational problem-solving

Each of that workflow's components, however, offers opportunity for elaboration. For example, the first step, "understanding the problem," is no trifle! In many cases, understanding the problem *is* the problem – what's more, it is a skill we can actively reinforce.

To that end, we have developed an institution-specific variation on CodingBat named Coding with Wally (Wally is our informal mascot)[1]. Figure 2's example displays Coding with Wally's interface; it is similar to CodingBat in that students are given the prompt, a code editor, and test cases. However, it

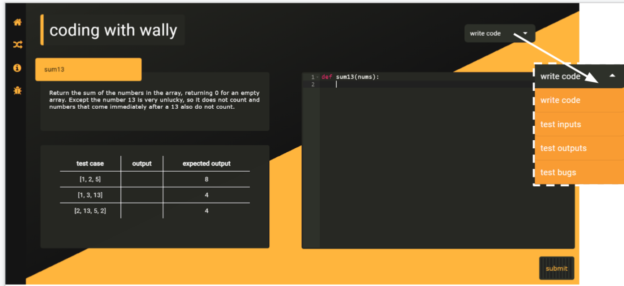

Figure 2: Coding with Wally augments CodingBat's approach, offering alternative ways of exploring a problem, its specification, and its input/output behavior. The interface supports the traditional approach of simply writing the function body. The upper-right menu offers several other options: predicting outputs, anticipating inputs, and finding bugs in provided, incorrect versions of the function body itself.

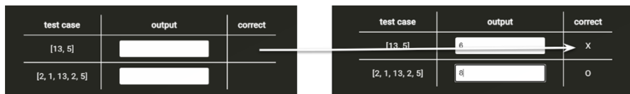

Figure 3: The test input section. Users are given test inputs and are asked to determine the corresponding output. The answer is checked, and the result shared.

differs in *also* offering each facet separately.

To focus on the challenge of internalizing what the problem "wants you to do," Coding with Wally offers several alternative interfaces.

Figures 3 and 4 show Coding with Wally's "test input" and "test output" pages of the previously-shown dropdown. Because it is more focused than the "wide-open" writing of a function body like in CodingBat, this mechanic is a powerful one for developing a deeper understanding of the problem. Before diving in, students have the opportunity to step back and carefully consider what the transformation should do, both forward and "in reverse."

This is especially useful for motivating edge cases and/or other details that may escape attention the first time the problem is encountered. It also reinforces that problem understanding is worthwhile – and takes time. Too often anxiety results from the inadvertently-absorbed belief that problem-understanding needs to be immediate: it shouldn't be!

A mechanic that further bolsters understanding is the "test bugs" section. A user is provided a buggy or incomplete version of the code, as shown in Fig. 5.

Figure 4: The test output section. Users are given test outputs and are asked to find a possible input (as there can be multiple). Again, the answer is checked and result shared.

Figure 5: The test bugs interface. Users are provided a buggy version of the code at right. The bottom left provides an interface for users to write inputs that reveal the bugs.

By parsing and digesting that code, students are asked to consider how it does – and doesn't – meet the problem specification. This interaction requires deep understanding with the problem specification and, more generally, *problem-specification space*! In most cases, comprehending code that someone else has written is more difficult than one's own code. This is all the more true when errors are lurking among its lines. (In this case, the buggy sum13 code does not appropriately consider the numbers that come *after* 13!)

Philosophically, the contributions of Coding with Wally echo the insights of Gamage's "Bottom-up" approach[2], in which entire, working artefacts provide a rich, authentic context. From that context, memorable and deeply engaging interactions arise. By approaching a computational challenge in multiple ways, each factored into a purpose-tuned interface, we conspire for each student to develop a nuanced and useful "computational grounding" from which to wrestle with future problems.

4 Slicer: Engaging a particular Python strength

At times, introductory students feel flooded with "sublanguages," those specialized pieces of every language whose expression is worth developing, first

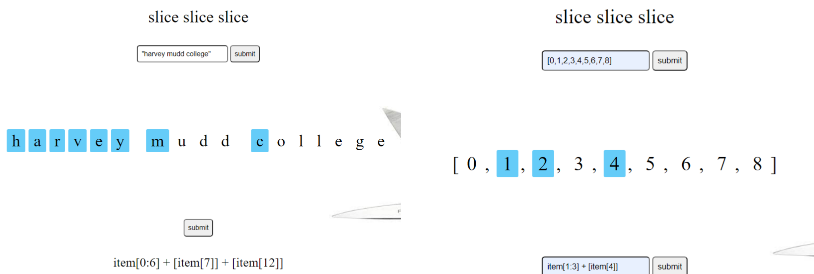

Figure 6: (**left**) Upon inputting a string or array into the first input box, the user is presented with that same input right below in large letters. The user can then "highlight" what they wish to keep by simply clicking on each index. Hitting the second submit button generates a possible combination of concatenated Python slicing to retrieve the indexes highlighted in blue. (**right**) Here, the user inputs their own slicing syntax. The website then highlights what part of the string or array that the user has selected with their code.

by practicing in isolation, then by integrating into a Coding-with-Wally-type challenge. (From there, it's on to problems "in the wild.") With Slicer, we scaffold this confidence-expanding process.

Python slicing and indexing can be difficult to grasp at first; it constitutes its own mini-language, complete with syntactic and semantic idiosyncrasies worthy of any full-fledged programming idiom! In general, slicing a sequence item takes the format of `item[start:end:stride]`, where `start` represents the starting index, `end` represents the ending index, and `stride` representing the number of indices traversed each step. For example, if `item="abcdef"`, then `item[1:4:2]` would evaluate to `"bd"`. There are many edge cases.

This syntax takes time for newcomers to digest. To help, we introduce *Slicer*, an interactive visual aid that singles out slicing syntax and invites students to build their own conceptual model mapping from that syntax to statement behavior[1]. Figures 6 and 7 show two "directions" for these interactions.

5 Hmmm with Wally: Making "the Machine" Accessible

High-level programming languages frame most of students' "Comp1" interactions. Base-two representation is also part of the experience. Our curriculum further includes a unit on assembly language. We feel assembly valuable-as-knowledge (all software "runs in assembly," after all). It is also further op-

Instruction	Description	Aliases
System instructions		
halt	Stop!	
read rX	Place user input in register rX	
write rX	Print contents of register rX	
nop	Do nothing	
Setting register data		
setn rX N	Set register rX equal to the integer N (-128 to +127)	
addn rX N	Add integer N (-128 to 127) to register rX	
copy rX rY	Set rX = rY	**mov**
Arithmetic		
add rX rY rZ	Set rX = rY + rZ	
sub rX rY rZ	Set rX = rY - rZ	
neg rX rY	Set rX = -rY	
mul rX rY rZ	Set rX = rY * rZ	
div rX rY rZ	Set rX = rY // rZ (integer division; rounds down; no remainder)	
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)	
Jumps!		
jumpn N	Set program counter to address N	
jumpr rX	Set program counter to address in rX	**jump**
jeqzn rX N	If rX == 0, then jump to line N	**jeqz**
jnezn rX N	If rX != 0, then jump to line N	**jnez**
jgtzn rX N	If rX > 0, then jump to line N	**jgtz**
jltzn rX N	If rX < 0, then jump to line N	**jltz**
calln rX N	Copy addr. of next instr. into rX and then jump to mem. addr. N	**call**
Interacting with memory (RAM)		
pushr rX rY	Store contents of register rX onto stack pointed to by reg. rY	
popr rX rY	Load contents of register rX from stack pointed to by reg. rY	
loadn rX N	Load register rX with the contents of memory address N	
storen rX N	Store contents of register rX into memory address N	
loadr rX rY	Load register rX with data from the address location held in reg. rY	**loadi, load**
storer rX rY	Store contents of register rX into memory address held in reg. rY	**storei, store**

```
1   00 read r1
2   01 setn r2 1
3   02 jeqzn r1 06
4   03 mul r2 r1 r2
5   04 addn r1 -1
6   05 jumpn 02
7   06 write r2
8   07 halt
9
```

```
TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

In [4]: run hmmm factorial.hmmm

----------------------
| ASSEMBLY SUCCESSFUL |
----------------------

0: 0000 0001 0000 0001        00 read r1
1: 0001 0010 0000 0001        01 setn r2 1
2: 1100 0001 0000 0110        02 jeqzn r1 06
3: 1000 0010 0001 0010        03 mul r2 r1 r2
4: 0101 0001 1111 1111        04 addn r1 -1
5: 1011 0000 0000 0010        05 jumpn 02
6: 0000 0010 0000 0010        06 write r2
7: 0000 0000 0000 0000        07 halt

Enter number (q to quit): 4
24
```

Figure 7: On the left is a description of each operation in the Hmmm assembly language. On the right is a sample program that takes an input integer and outputs its factorial. This also shows how students traditionally run Hmmm programs through the terminal.

portunity to practice "computational patterns,"[4], i.e., conceptual models of computing processes. There is a cohort-building facet, too, borrowing the spirit of experiments such as [5].

A short, hands-on tour of assembly uncovers a layer of abstraction that enriches the experience of high-level program development, and opens doors to fuller models when the need arises. In a way, assembly is computing's "genetic translation and transcription." Like genetics, it's worth having as part of a computational worldview – even if students don't see their future selves wrestling with machine architectures (or biological ones!)

6 Hmmm...

Thus, every student programs in the assembly language, Hmmm, in their required computing course. Hmmm, the Harvey Mudd miniature machine, is a small, conceptually central subset of all in vivo assemblies. Hmmm digestibly conveys instruction syntax and direct interaction with registers and memory. The machine itself is a 16-register system with 256-memory locations, emulated by a single Python file. Students run, debug, and reason about the Hmmm code they write.

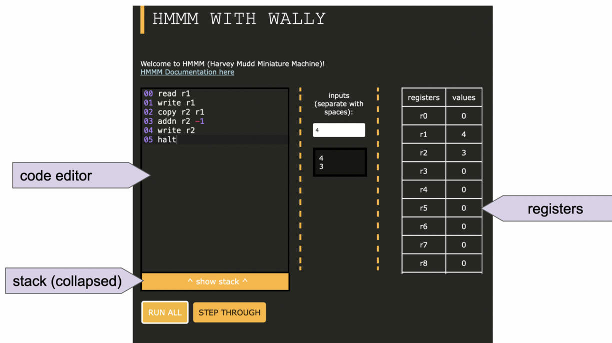

Figure 8: App interface: The page has a simple layout, displaying the editor at left, registers at right, and input/output boxes in the center. The short program shown in the editor was run with the "run all" button. This is similar to how students would previously run their code, with the added benefit of seeing the state of the registers and stack at the end of the execution.

7 Accessible Assembly: Hmmm with Wally

Though Hmmm makes the low-level mindset accessible, all assembly language can be chin-scratching stuff! Beyond the command-line interface, we present here an accessible web application with which students can visualize and tinker with what their Hmmm code is actually doing – and how. This interface reinforces the conceptual model we hope all students take away from their Hmmm experience. Figures 8, 9, and 10 show this interface and summarize its opportunities.

8 Perspective(s)

Sandboxes – where students can focus on one facet of a computational model – offer benefits especially when computing is a universal, shared experience. This work has illustrated the vision – and advantages – of embracing many exploratory and explanatory paths of computing-as-literacy. When building confidence and comfort with a new mindspace, multiple approaches – unpacking problems from different perspectives – offer "onramps" into engagement and understanding. For skills as broadly applicable as computing, this is all the more important. As computing embraces more roles, such approaches are vital: they open doors in all directions, both inward to further computational work and onward across disciplinary specialties.

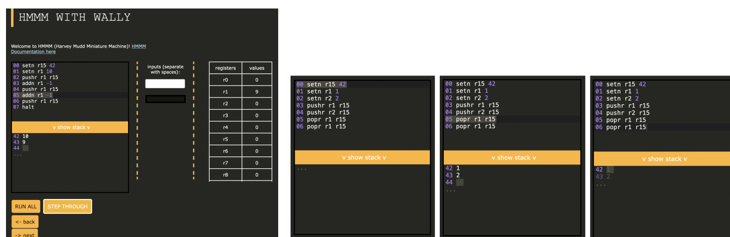

Figure 9: (**left**) "Step through" functionality allows students to run one instruction at a time, moving forward and back incrementally through their code, seeing how registers and the stack changes with each line. Here, the last three lines of code have not yet been run, as reflected in the stack and registers. (**right**) The stack is positioned and changed as an extension of the editor to emphasize how both instructions and the data-stack are stored in the same set of memory locations.

Acknowledgments The authors gratefully acknowledge the funding support of NSF projects 1707538 and 1612451, along with resources made available by Harvey Mudd College.

References

[1] *All of the authors' applications in this work are available at https://myappkanalu.firebaseapp.com/.*

[2] Lasanthi N. Gamage. "A bottom-up approach for computer programming education". In: *Journal of Computing Sciences in Colleges* 36 (April 2021), pp. 66–75.

[3] Nick Parlante. "Nifty reflections". In: *SIGCSE Bulletin* 39 (2007), pp. 25–26. DOI: 10.1145/1272848.1272876.

[4] Richard Rasala and Viera K. Proulx. "Pattern and toolkits in introductory CS courses". In: *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference* (October 2000).

[5] Rita Sperry. "We're all in this together: learning communities for first-year computer science majors". In: *The Journal of Computing Sciences in Colleges and Proceedings of the 32nd Annual CCSC South Central Conference* (April 2021), pp. 11–19.