Design and Evaluation of a Block-based Environment with a Data Science Context

Austin Cory Bart, *Member, IEEE,* Javier Tibau, *Member, IEEE,* Dennis Kafura, *Member, IEEE,* Clifford A. Shaffer, *Senior Member, IEEE,* and Eli Tilevich, *Senior Member, IEEE*

Abstract—As computing becomes pervasive across fields, introductory computing curricula needs new tools to motivate and educate the influx of learners with little prior background and divergent goals. We seek to improve curricula by enriching it with authentic, real-world contexts and powerful scaffolds that can guide learners to success using automated tools, thereby reducing the strain on limited human instructional resources. To address these issues, we have created the BlockPy programming environment, a web-based, open-access, open-source platform for introductory computing students (https://www.blockpy.com). BlockPy has an embedded data science context that allows learners to connect the educational content with real-world scenarios through meaningful problems. The environment is block-based and gives guiding feedback to learners as they complete problems, but also mediates transfer to more sophisticated programming environments by supporting bidirectional, seamless transitions between block and text programming. Although it can be used as a stand-alone application, the environment has first-class support for the latest Learning Tools Interoperability standards, so that instructors can embed the environment directly within their Learning Management System. In this paper, we describe interesting design issues that we encountered during the development of BlockPy, an evaluation of the environment from fine-grained logs, and our future plans for the environment.

Index Terms—Computer Science Education, Introductory Computing, Block-based Programming, Data Science, Automatic Guidance.

1 Introduction

As computing has become pervasive across careers and disciplines, there is a growing population of students and professionals alike seeking to develop computational skills and thought processes [1]. Efforts to address their needs include general education curricula in higher and secondary education (e.g., "Computational Thinking" and "Computer Science Principles" courses) [1], [2], Massive Open Online Courses [3], and even individualized, informal learning platforms (e.g., CodeCademy) [4]. Because these learners have dissimilar motivations, clarity of goals, and depth of prior experiences [5], [6], they need support from educational different from that provided to traditional Computer Science students [7], [8].

We seek to support these learners by means of BlockPy, a web-based, open-source, introductory programming environment (https://www.blockpy.com), emphasizing data science as an authentic learning context. By an authentic context, we mean that learners perceive a connection to a real-world community of practice that is relevant to their long-term goals. The environment provides scaffolds through a dual text/block interface and guiding feedback, making it well-suited for introductory computing students, particularly non-majors. Although BlockPy may be sufficient for some learners, the system can also be a first step toward a more professional environment.

This paper presents a comprehensive description and motivation for key features of BlockPy. This paper also continues our evaluation of BlockPy's design and features. The positive impact of BlockPy's data science context on student motivation has been studied in our prior research [9]. This paper makes the following new contributions:

- Description of BlockPy's major features and rationales for educators interested in learning environments for novice programmers, especially those considering a block-based approach.
- Reflection on design issues involved in BlockPy for developers in the block-based community who would wish to build systems similar to BlockPy.
- Evaluation of BlockPy's features, scaffolding, and guided feedback for adopters and developers.

- *Computer Science, College of Engineering, Virginia Tech, Blacksburg, VA, USA.*
  Corresponding Author: Austin Cory Bart (acbart@vt.edu)
- *This material is based upon work supported by a grant from the National Science Foundation Graduate Research Fellowship, Grant No. DGE 0822220, NSF DUE 1444094, and NSF IUSE 1624320.*

Manuscript received May 1, 2015.

We begin in Section 2 by placing BlockPy in relation to other block-based languages and other introductory programming contexts. Section 3 presents the material for our first contribution by describing eight of BlockPy's core features. The high-level implementation of some of these features are discussed briefly. Reflections on design issues, our second contribution, is presented in Section 4. These reflections highlight several challenges or trade-offs involved in the creation of BlockPy. Section 5 explains the methodology used in the evaluation, which involves both qualitative and quantitative components, that answer these three research questions:

1) Do novice learners find the BlockPy features helpful?
2) Is the BlockPy scaffolding effective in transitioning novice learners from a block-based environment to a professional text environment?
3) Is the guided feedback effective and sufficient for novice learners?

A roadmap for BlockPy's future development is given in Section 6.

2 The *Why* of BlockPy

BlockPy draws inspiration from a number of theoretical and concrete sources. Design decisions were influenced by educational theories, existing introductory programming environments, and the wider community of professional software developers.

2.1 Python and Blocks

BlockPy is a block-based editor for the Python programming language. Python is a popular introductory language because of its beginner-friendly syntax, powerful libraries, and popularity among professional programmers [10]. So not only are novice learners likely to be successful when initially working with Python, but they can quickly enter into an authentic community of practice and solve real-world problems. Python also has a well-earned reputation as a language for performing data science [11], ensuring a harmonious relationship between BlockPy's language and context, described in Section 2.2.

Block-based languages have proven themselves as a powerful scaffold for novice learners, decreasing their start-up time and helping them accomplish tasks that they otherwise could not [12], [13]. Blocks help beginners navigate their program's structure while preventing syntax errors. Blocks can also visually and clearly expose a complex API, such as those used in game development or data processing. These benefits offset a major disadvantage of blocks: learners can negatively perceive block interfaces as being only for younger learners or unsuitable for professionals [14].

BlockPy has much in common with other block-based programming environments. For example, Scratch and Snap! [15] largely target young learners, both in design and with their game development context. Extensions to Snap! have integrated more sophisticated features, although these have been limited. Hellman [16] incorporated data science features, including access to real-time datasets, user-created data sources, and cloud-based data manipulation. The NetsBlox project [17] exposes distributed computing concepts by introducing event blocks for network transmissions. Others have promoted patterns for parallel programming abstractions within Snap!, such as producer-consumer and MapReduce [18].

BlockPy's emphasis on authenticity is similar to that of GP, the "The Extensible Portable General Purpose Block Language for Casual Programmers", which seeks to support more ambitious application development. Developed by members of the Scratch team, GP shared many of its design principles, including the concept of a strong social community and a blocks-first interface. A unique aspect of GP is that its development environment and module system is extensible with its own internal block-based language. The GP project attempts to establish authenticity by supporting real-world features and projects that "scale up" [19]. A potential drawback of this approach to authenticity is that instead of an existing popular language a language descended from Squeak is used.

BlockPy has much in common with other modern block-based editors for mainstream languages. PencilCode is a JavaScript editor that offers a seamless transition between blocks and text [20]. GreenFoot is a visual programming environment for creating games and animations in Java, with an innovative structured code editing interface they refer to as "Frames" [21]. These systems suggest that supporting a dual-interface between blocks and text is a trend in modern editors as a mean of transitioning students gracefully between different programming editors.

BlockPy is not the first web-based Python execution environment, but it advances the state-of-the-art established by its predecessors, including Pythy [22], CodeSkulptor [23], and the Online Python Tutor [24]. None of these systems support a dual block/text interface. Both CodeSkulptor and Pythy are built on the same underlying engine, Skulpt [25], which can cross-compile Python code to JavaScript. CodeSkulptor has an extensive but custom API for creating user interfaces and games, which is powerful but limits students' ability to transfer code away from the browser. CodeSkulptor is intended as an undirected environment for creativity, but therefore does not guide learners through a curriculum. Pythy, on the other hand, is an assignment-oriented application with limited support for guidance through unit testing. Pythy appears

to no longer be under active development, and uses an out-of-date fork of Skulpt.

The Online Python Tutor (OPT) uses remote code execution to provide visualizations of users' algorithms. Although rigorous and detailed, OPT requires learners to master the complicated terminology being used. Further, OPT is an undirected environment like CodeSkulptor, rather than a platform for a curriculum. Finally, its dependency on a remote server makes the platform vulnerable to poor internet connections and complicates the applications' architecture.

In addition to its dual text/block interface, BlockPy also provides guided feedback. Guided feedback is well-known as a powerful mechanism for improving student learning [26]. There are a wide number of techniques and tools that can be used to provide various levels of guided feedback to learners [27]. These techniques include unit testing, static and dynamic analysis, and even more complex processes such as programmatic transformations. BlockPy provides tools to the problem designer to give customized feedback. These tools are inspired by systems such as Ask-ELLE [28], where the instructor can demand certain characteristics of the program and specify feedback.

2.2 Data Science as an Authentic Context

The design of BlockPy was influenced by two educational theories, both of which stress the importance of a learning experience that students can connect to the real world. The first theory is the MUSIC Model of Academic Motivation [29]. This meta-descriptive theory, which aggregates other motivational theories, draws a distinction between students' perception of Usefulness (a sense that the material is connected to long-term goals) and Interest (a sense of intrigue and inspiration). Appealing to a student's sense of usefulness or interest is argued to strengthen their motivation, not only to complete individual assignments, but to engage with computing and understand its role in their long-term career and life goals [5]. The second theory is Situated Learning Theory. This theory describes how learners become engaged with material when they can perceive its authenticity and connection to a community of practice that they might reasonably become a part of [30]. The theory distinguishes between the content (the material to be learned), the scaffolds (the artificial additions to the environment to simplify the students' experience), and the context (the framing story used to anchor the experience). A context is authentic when the problems, practices, tools, artifacts, and/or results that students work with in their learning reflect real-world entities described by authoritative sources in a community of practice.

Many introductory computing contexts focus on invoking student interest, without providing a sense of usefulness or authenticity. Media Computation, for instance, is a creativity-based curriculum by Guzdial et al. where students create art and music. Although there is established motivational value in this approach [31], an analysis by Guzdial in the light of Situated Learning Theory suggested that, despite extensive efforts by course staff, students did not perceive the context as authentic or useful [32], in part because students did not perceive the context to align with a visible community of practice. Game Design is another alternative context that does have a visible community, but arguably not many students seek to join that community as part of their career plans. We do not believe that all introductory programming environments must be authentic and focus on usefulness. For young learners, appealing to their sense of play or story-telling might be more appropriate. We do believe that for our learners (university non-majors), authenticity and usefulness are critical.

BlockPy is part of a growing movement within computer science education to promote "data science" as an authentic context appealing to the usefulness of students in any discipline or career path. The argument is that every job and major, from the sciences to the humanities to the arts, can benefit from the ability to solve problems from a data-oriented computational perspective. Data science is authentic because 1) the data is from authoritative sources, 2) there is a real-world community of practice for data science, and 3) students can more readily see how the material connects to their long-term career goals.

There are several prior data science curricula. The authors had prior success in using a data science context through CORGIS, the Collection of Real-time, Giant, Interesting dataSets, which makes it easy to add real-world datasets into an introductory computing curriculum. This project has been deployed in a college-level Computational Thinking course, with great effect on student motivation [9], [33]. Anderson et al. suggested a curriculum built around real-world projects and data sources [34]. Goldweber et al. [7] developed an entire framework for evaluating and designing projects with this theme. The BRIDGES project [35] has a similar goal to the CORGIS project, but is targeted at upper-level Data Structures and Algorithms courses, and has a stronger emphasis on data visualization and exploration.

BlockPy facilitates a data science context, and provides tools for students to rapidly begin working with real-world datasets relevant to their personal and work interests. Note that BlockPy, as a programming environment, does not provide a sense of authenticity. Instead, it facilitates a context that does. The computing curriculum and pedagogical decisions surrounding BlockPy are heavily influenced by this context. Lessons are built around collection-based iteration, for

instance, as opposed to conditional iteration, which more naturally connects to working with collections of real-world data. Tools are also provided by the environment for conveniently visualizing and manipulating data. BlockPy is designed as an active learning environment, with an emphasis on students interacting and receiving feedback. This means that BlockPy is designed to require minimal amounts of instruction and presentation of material. At the same time, a major theme of BlockPy is scaffolding—pedagogical and technological support that enables the learner to accomplish tasks they normally would be unable to achieve. As the learner becomes more capable, the scaffolding fades away.

3 *BlockPy*'s Major Features

In this section, we describe the major features of BlockPy. We view the contribution of BlockPy as the synthesis of known features that are carefully integrated into an environment of novice learners, within the technical constraints of client-side execution. An overview of the web-based interface is shown in Figure 1. At a high level, the left side of the interface is the editor, and the right side is where code execution and output is visualized.

At a high level, the left side of the interface is the editor, and the right side is where code execution and output is visualized. The editor area has a menu on the left from which blocks can be selected and a canvas for composing the blocks to create a program. The menu items beginning with "Data" expose blocks for accessing data in the CORGIS library. The editor area also contains a row of buttons. The "Run" button initiates a browser-based execution of the program. The "Text" button controls the dual block/text modes. The "Pseudo" button displays a pop-up window with a natural language description of the program. Immediately above the row of buttons is the "Feedback" label next to which guided feedback is displayed. In Figure 1, the guided feedback is the word "Success" indicating that the program is correct according to the instructor-authored checks for this problem. The area on the right labeled "Printer" displays the program's output. In Figure 1, a line graph produced by the program is shown. The buttons at the top ("First", "Back", etc.) control the property explorer whose information is displayed in the "Trace table" at the bottom.

Dual Block/Text Modes One of the most visible features of BlockPy is its dual text/block interface. At any time, users can switch between a block or a text representation of their code, as shown in Figure 2. The dual block/text mode is an important scaffold, gradually allowing the learner to move from a supportive block environment to a more professional text-based programming environment. Our experience with university-level non-computer science majors is that scaffolding of this kind is important. The block interface uses the Google Blockly library [36], while the text interface uses the CodeMirror library [37]. The Blockly library has been extended with a number of new blocks and features to connect more gracefully with the Skulpt execution engine. In particular, Skulpt parse trees can be converted into Blockly parse trees.

Guided Feedback Another valuable pedagogical component of BlockPy is its Guided Feedback system. As shown in Figure 3, instructors define a function taking the students code, a trace of its execution, and any output produced. When the program is run the function is automatically executed and can return feedback or an indication that the output is correct. By combining static and run-time information the feedback code can provide many kinds of guidance: suggestions on what is wrong, hints on what to fix, helpful readings, or even complimentary feedback. For example, in a problem to sum a list of integers, a student could be informed that the iteration statement is not properly formed, that an initialization is missing, or that a given section of the textbook is relevant. Complimentary feedback can sustain the student's motivation and help avoid misguided changes to correct parts of the program. The desire of students for informative feedback and the extensible nature of the feedback mechanism motivates improvements described in the future work.

Python Execution Environment BlockPy uses the Skulpt engine to execute Python code entirely within the users' browser. Skulpt works as a "transpiler", or source-to-source compiler. It parses a string of valid Python source code into an Abstract Syntax Tree represented as a JSON-encoded object. A symbol table is constructed, and then a JavaScript execution engine interprets the AST. No bytecode is created, and the JavaScript is executed within the client's browser. Skulpt uses suspensions so that code execution is a non-blocking activity.

The biggest advantage of this approach is that code can be executed much faster since there is no round trip to a server. Students can continue to work without an Internet connection. Complicated sandboxes are unnecessary for running the student's code, since they are limited to the API exposed by their browser. In fact, Skulpt even protects the client's environment, since the Window namespace is hidden.

Natural Language Code Description Another scaffold of BlockPy is a natural language program description generator. Conventionally, written code is parsed into an Abstract Syntax Tree. In BlockPy, the transition occurs in the other direction—an AST is used to generate a string of conventional English text. It can be seen as a third phase to the existing dual text/block

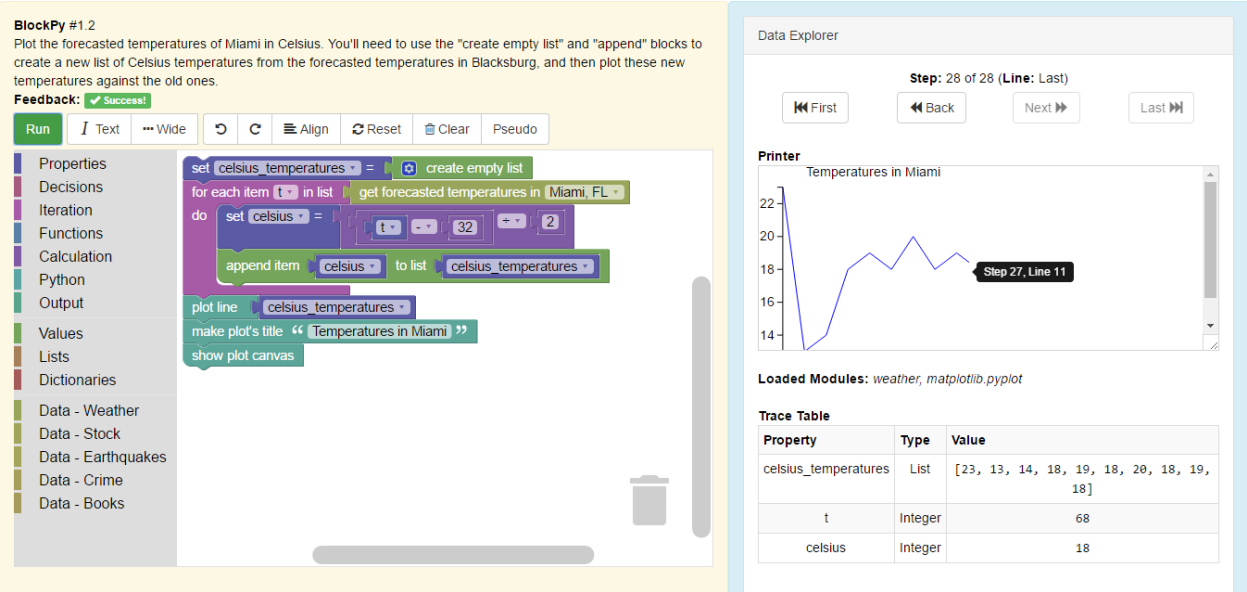

Fig. 1. A Screenshot of BlockPy in Action

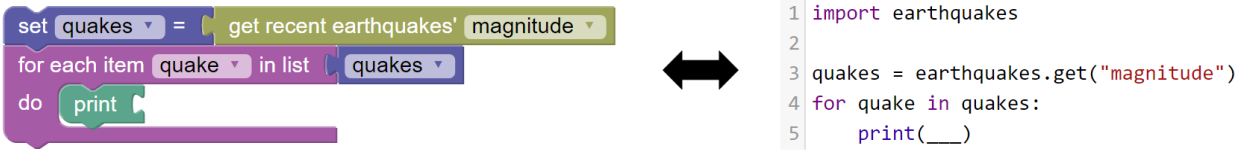

Fig. 2. Mutual Language Translation between Block and Text interfaces

```
def on_run(code, output, trace):
    if not output:
        return "Not Printing!"
    if "for" not in code:
        return "Need a FOR loop!"
    if 'sum' not in trace:
        return 'Need a sum variable!'
    if output == 42:
        return 'Success!'
    return 'Keep at it!'
```

Fig. 3. Example of Instructor-written Guided Feedback code

mode, although it does not support editing, being a one-way transition. The goal of this code explanation is for students to better understand the meaning of their code. Obtuse language features can be translated into more meaningful statements.

Property Explorer After a program is run, BlockPy supports traversal of the executions' trace. We have found through classroom observations that introductory students often struggle to trace the execution of their programs on their own. Using the property explorer, not only can students observe the appearance and value of their variables, but also their type. Further, they can "rewind" print and plot operations to observe the impact of these statements.

CORGIS Integration BlockPy promotes a data science learning context by natively integrating CORGIS. CORGIS contains diverse data, from authoritative sources. The subset of CORGIS exposed in the current BlockPy interface is selected based on perceived popular appeal, simplicity, and pedagogical affordances of the data. These libraries are available through simple blocks. These blocks translate into function calls that return structured data at varying levels of complexity depending on the block chosen (e.g., `get_temperature` returns a single integer, `get_temperatures` returns a list of integers, `get_forecasts` returns a list of dictionaries with integers and strings, etc). Figures 1 and 2 demonstrate these blocks in use.

Plotting Another addition to Skulpt is tools for making visualizations, a core activity for data science. Currently, BlockPy supports the creation of line plots, scatter plots, and histograms using an API identical to the popular MatPlotLib API.By mimicking this professional API, BlockPy increases authenticity and promotes transfer. Students in our course, for example, transfer this learning to their final projects where they use MatPlotLib from the Python program they write.

LTI Support Although BlockPy has its own internal course management system, it also supports

LTI (Learning Technology Interoperability). That is, it provides a mechanism for Learning Management Systems (e.g., Canvas, Sakai, Blackboard, Moodle) to interact with external tools. Instructors who have configured BlockPy to work with their LMS can select, create, and edit assignments all from within their LMS of choice. When students complete problems in the BlockPy environment, they are automatically graded, and the LMS is given the relevant information.

LTI supports user authentication through Canvas (and potentially other LMS). When a student loads a BlockPy problem, Canvas delivers their email address to BlockPy. This information is used to determine if an account exists for that student. If it does not, they are introduced to the system and vouched for by the LMS.

4 Design Issues

We next present design issues that were encountered when developing BlockPy. These issues should be of interest to developers of introductory environments.

4.1 Internal Code Representation

Dual block/text editors create an added challenge in deciding the right internal representation of the students' code. BlockPy uses the textual version as the canonical representation, as opposed to the block parse tree. This choice facilitates the text mode display and program execution. However, there were other options. Some editors operate on Parse Trees exclusively. Others treat the source code as a list of lines (separating elements by the newline), sometimes attaching special properties to individual lines such as geometric information [20]. A major limitation of both representations is that some valuable programmer-level semantic data is not preserved. On the text side, user-created whitespace does not survive the transition. On the block side, block layouts are reset according to the default rules. The original dual text/block editor created by Mastuwaza [38] solved this problem by storing geometric information of blocks in the comments of the text mode. However, this leads to crowded code with confusing comments. The trade-offs in Mastuwaza's system led to BlockPy's design as a primarily text-driven environment.

4.2 Block Language

A criticism of the block interface is that the blocks do not use accurate Python syntax. For example, the collection iteration block that models a Python `for ... in` loop has plain text phrasing, to explain the nature of the block more clearly: `for each item [__] in list [__]`. Similarly, the assignment block has the text `set [__] = [__]`. Over time, BlockPy's exact wording has evolved to more closely match Python. However, we feel that explicit text is more helpful to beginners. Although there are advantages to more understandable blocks, there are credible concerns that beginners may learn incorrect syntax. We comment further on this in Section 5.3.

4.3 Parser Errors vs. Syntax Errors

In theory, it is impossible to generate syntactically incorrect Python code when transitioning from the blocks to text. However, it is quite possible for students to write invalid code in the text mode, making the transition back to blocks problematic. A missing colon, unclosed parentheses, or incorrect indentation will prevent Skulpt from generating a valid parse tree. When encountering code with a syntax error, BlockPy creates "raw blocks" that store the literal Python code. BlockPy will also create raw blocks for language features that are not implemented in the block interface, but most of these features are uncommon, such as `else` bodies in `for` loops.

The algorithm for translating code attempts to create as many blocks as possible, but can often be confounded into creating one large raw block. Although some might consider this a disadvantage, it is not necessarily desirable to ensure that students always write completely valid programs at all times, especially in the early stages of constructing an algorithm or when new programming constructs are introduced. Although BlockPy is built on the premise that it is worth delaying the conversation about syntax, students must eventually become comfortable with the details of writing syntactically correct text code.

It is not always possible to automatically correct a students' written code to match their intent. (Sometimes students may not even understand their own intent!) However, there are more sophisticated approaches to improving the support given to students. A more robust parser could be developed to precisely identify student code errors. Alternatively, every subset of the code could be parsed in isolation in order to determine what areas of the code are correct.

5 Evaluation of BlockPy

We now describe the results of a deployment of BlockPy. BlockPy has been used in an introductory Computational Thinking course for four semesters. This course is meant for non-Computer Science majors from the humanities, arts, and the sciences. They typically have no prior programming experience and have a limited understanding of the field. Therefore, they ideally model the anticipated BlockPy user.

5.1 Methodology

In the Spring 2016 offering, 50 students were assigned 34 BlockPy problems over the course of 4 classes

TABLE 1
Overview of the BlockPy Curriculum

Problems	Type	Topics
#1.1-#1.5	Classwork	Printing, Variables, Plotting
#1.6-#1.8	Homework	Printing, Variables, Plotting
#2.1-#2.5	Classwork	Iteration, Accumulation, Mapping
#2.6-#2.8	Homework	Iteration, Accumulation, Mapping
#3.1-#3.6	Classwork	Conditionals, Filtering Lists
#3.7-#3.8	Homework	Conditionals, Filtering Lists
#4.1-#4.6	Classwork	Textual Code, List Transformation
#4.7-#4.A	Homework	Textual Code, List Transformation

over 2 weeks (see Table 1). This focused use of block programming built upon their notions of algorithms, which they had previously seen during a 6-week introductory unit where students created algorithms using natural language and flowcharts. Most of the problems were assigned as classwork, with the expectation that any incomplete assignments were to be done as homework. Typically, classwork problems would give students starting code. Homework questions would have them complete similar problems from scratch. After the BlockPy unit, the curriculum continued in the Spyder IDE, a more conventional desktop programming environment. To facilitate this transition, the last day of BlockPy material started in text mode, and encouraged students to become familiar with writing code in that form. BlockPy was not intended to be used for all programming in our course, and was faded in the transition to Python. In contrast to other curricula [39], for our college-level learners, the BlockPy scaffolding could be removed after 4 days. More information about the curriculum can be found in [33], and at the course's public site [1].

Figure 4 describes student completion rates over the assignments. The bars are colored to indicate day (there were four days of BlockPy activities), and their brightness indicates homework vs. classwork. Figure 5 describes the distribution of time spent on each problem. 92% of the 50 students completed more than 60% of the 34 problems and 62% of the students completed more than 90% of the problems. The average student was able to complete most problems in 15 minutes, which was considered reasonable by the instructors.

BlockPy evaluation is based on two data sources. First, fine-grained logs were collected of the students' interactions with BlockPy, including any changes made to their code at the keystroke/block level and interface actions. Second, a survey was administered 2 weeks after BlockPy was completed. 41 out of 50 students gave consent for their answers to be released for research purposes, with 19 male students and

1. https://think.cs.vt.edu/course_materials/

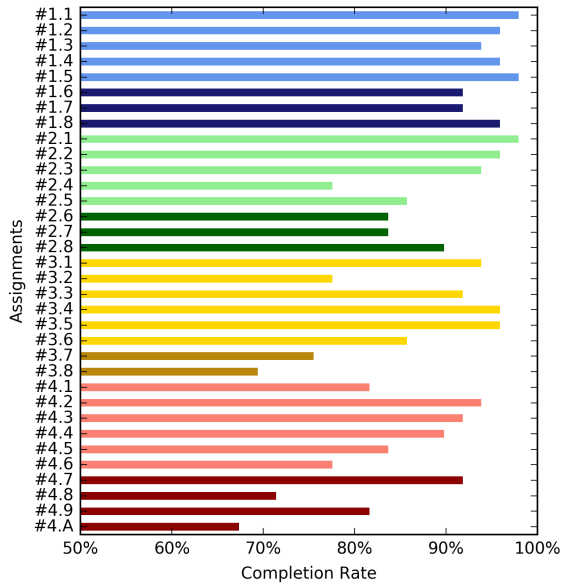

Fig. 4. Completion Rate by Problem (colors indicate distinct days, brightness indicates homework vs classwork)

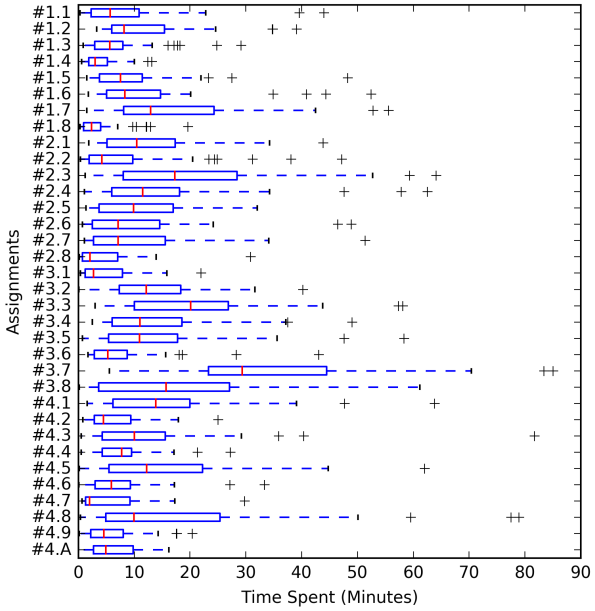

Fig. 5. Work Session Length (minutes) by Problem

22 female students. 32% were freshmen, 39% were sophomores, 17% were juniors, and 12% were seniors.

This survey had four free-response questions asking students about frustrating and helpful features in both BlockPy and Spyder:

1) What features of BlockPy were most helpful?
2) What features of BlockPy were most frustrating?
3) What features of Spyder were most helpful?

TABLE 2
BlockPy Survey Qualitative Results

Features	Generalized Tag	N (%)
Helpful BlockPy	Block Interface	24 (57.1%)
	Dual Text/Block	12 (28.6%)
	Guidance	5 (11.9%)
Frustrating BlockPy	Vague Guidance	19 (45.2%)
	Block Interface	7 (16.7%)
	Programming	5 (11.9%)
Helpful Spyder	Better errors	16 (38.1%)
	Variable Explorer	17 (40.5%)
	Writing text code	5 (11.9%)
Frustrating Spyder	Error Messages	12 (28.6%)
	Text Coding	7 (16.6%)
	Interface	6 (14.3%)
	Crashing	5 (11.9%)

4) What features of Spyder were most frustrating?

We performed qualitative, open coding on the free response questions, converging on a set of generalized tags. 42 students responded to each of the 4 questions. The responses were brief phrases and sentences, with only a scant few having more than one sentence. One author was tasked with performing the open-coding on the survey responses in order to find general tags; each response was first condensed into a brief specific tag, and then common tags were grouped into more general clusters. For example, for the Frustrating BlockPy features question, there were 42 responses which were generalized into 21 specific tags, which were in turn generalized further into 10 tags. Table 2 presents the generalized tags gathered during the qualitative coding. The tags are grouped by the Features column, which identifies which environment students were responding to and whether they were considering its helpful or frustrating aspects. The number of occurrences of each tag across student responses is given in the third column. Only tags with 5+ occurrences were included. Sections 5.2 and 5.4 describe the results of this analysis.

The survey asked this multiple-choice question about where students felt the BlockPy/Spyder transition should occur: "When do you think we should have STOPPED using BlockPy and started using Spyder?" In addition to an "Other" response, six alternatives were given: the three choices "Always", "Never", "The Same", and and three different places in the curriculum, implying more or less use of BlockPy. The results of this question are discussed in Section 5.3.

5.2 BlockPy's Helpfulness

Do novice learners find the BlockPy features helpful?

To answer this primary research question, the results of the survey were analyzed for commonly repeated sentiments by students. Students gave positive responses about BlockPy's block interface. 57% indicated the block interface as particularly helpful. Two representative responses are:

"You could much more easily visualize the code, and you didn't have to worry about grammar and the text language."

"That we didn't have to write the code and worry about indentation, punctuation."

29% indicated that the dual text/block interface helped. For example:

"I liked the ability to switch between text code and the blocks"

"Being able to switch from text to block view. Even after we stopped using BlockPy I could put my code into it and it helped me [see] where I was wrong"

Taken together, this feedback, comprising 85% of the students, suggests that the highly visible block interface is what students find most helpful in BlockPy.

Only 12% found that the guidance offered by the feedback mechanism particularly helpful:

"... I found it helpful when there was information to guide me when I could not proceed and obtain the correct answer."

"... I also liked that it would give hints and tell you when the code was correct.... "

In terms of frustrations, 19% of the students found the guidance provided by the automatic feedback mechanisms to be vague. This issue is discussed further in Section 5.4. While the block interface was generally well received,

"Making the blocks link up in the way I wanted was kind of finicky sometimes."

"the constant dragging of blocks."

Other frustrations were with the problems assigned or the nature of coding in general.

"everything when you can't figure out how to successful run the program"

Helpful features of Spyder included better error reporting and the variable explorer feature. While the feedback in BlockPy was criticized by 45% of the students, 38% found the feedback in Spyder helpful. While both BlockPy and Spyder included a variable explorer feature (to display the values of variables during and after execution), this feature was mentioned positively by 40% of the students. The positive view of this feature in Spyder may be due the greater need for a variable explorer in the text programming part of the course which introduced more complex data structures. Comparatively, many criticisms of Spyder related to coding in general, rather than features of the environment (e.g., students described frustrations with Python syntax, trying to interpret errors).

"Very small details that are inherent in coding. Things like capitalization and other very small details were annoying..."

"None. Writing code in general is frustrating."

5.3 Evaluation of the Scaffolds

Is the BlockPy scaffolding effective in transitioning novice learners from a block-based environment to a professional text environment?

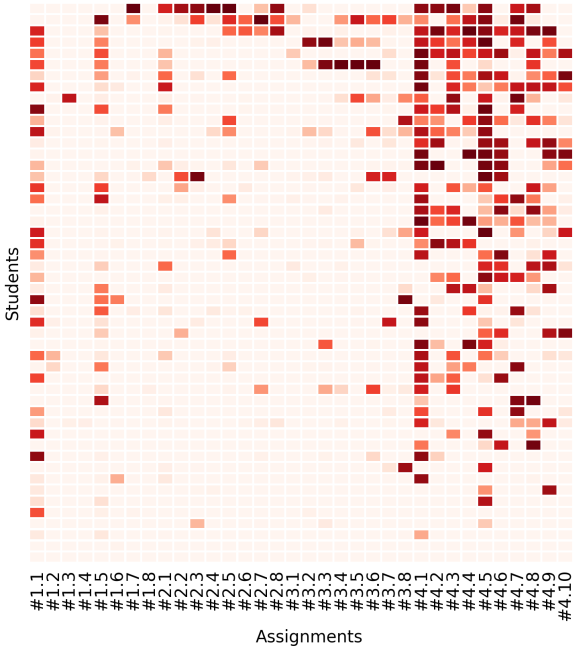

Fig. 6. Time Spent in Text-mode by Students on each Assignment

To answer this question we analyzed various aspects of the students use of the dual block/text feature of BlockPy, particularly during the transition from BlockPy to the text-only Spyder IDE. Figure 6 is a grid-based representation of students' time spent in text mode while working on assignments, with redder cells indicating more time spent in text mode. Each row of cells represents another students' progression through the assignments, given on the X-axis. For the first 3 days of the assignments, students mostly stayed in the block interface (with the first problem being a notable exception, since it was used to show students how the interface was dual-block/text). It is encouraging to note that some students did switch between the block and text editor in several early problems, if only to observe the resulting code. However, it is also clear that many students stayed almost entirely in the block mode, even during the final problems. This graph does suggest that more incentives and guidance could be given to direct students to pay attention or take advantage of the text interface when the problem expects them to, to build their competency with that form of their programs. It is interesting to note that there are a small number of students who used the text mode almost exclusively; anecdotally, some students expressed that they found the text interface more understandable than the block interface. It is another advantage of the dual block/text interface that both kinds of students, those who prefer blocks and those who prefer text, can use the system without issue.

The last day of BlockPy began the transition to full textual Python programming. Students tended to spend more of the last day in the text interface, which was encouraged by the interface starting in text mode and the problems asking students to write their programs in text rather than blocks where possible. To prepare students for the transition, they are given a translation guide showing individual blocks and their equivalent textual form in Python. The day's lecture covered basic text syntax, and gave side-by-side examples of blocks versus Python text.

One effect observed in the transition to text was that students were confused by the difference in keywords between BlockPy and Python. As previously described, the keyword text on blocks is often more verbose than the actual Python syntax. This is intended as a feature to improve learners' understanding of the blocks. However, analysis of the logs suggest that this causes confusion for some students when they transition to writing text code. A crucial question is how wide-spread this problem is and how long it persists. We looked at three types of mistakes that students could make due to the keyword differences: writing `for each X in Y` instead of `for X in Y`, writing `set X = ...` instead of `X = ...`, and writing `if X then do ...` instead of `if X`. We found 20 students who exhibited this behavior in 29 different incidences across all the problems: 9 incorrect `for`, 17 incorrect `set`, and 1 incorrect `if`. Few students made the mistake on more than one problem. In over 60% of the cases, the student corrected their mistake within a minute, and in 23% the student corrected their mistake in 1-4 minutes. In the two worst cases, students persisted with the incorrect code for over 5 minutes and 7 minutes. It is worth pointing out that no student ended the assignment (successfully or not) with any use of the word `each`, `set`, or `do`. While the degree and persistence of the keyword confusion problem is very limited, it does seem prudent for the guidance to correct students who might be using such incorrect code forms.

The survey asked students where they would prefer to make the transition from BlockPy to Spyder, and allowed to choose from a series of intervals in the course. The majority (28 students) indicated that the current location was appropriate, 6 students indicated that it should be later, 6 indicated that instruction should be kept in BlockPy, and 7 students indicated that it should be earlier. This provides justification for the relatively short duration of the BlockPy curriculum compared to subsequent sections of the course—the curriculum is intended to transition students into a

```
import stocks

stocks = stocks.get_past("FB")
new = []
for stock in new:
    new = []
for stocks in stocks:
    print(stocks)
```

Fig. 7. Degenerate Student Code

more mature environment, and so is not meant to last for too much of the course.

5.4 Feedback Quality

Is the guided feedback effective and sufficient for novice learners?

To answer this question the free response questions were analyzed. There was little agreement about what was most frustrating about BlockPy, with one major exception: 45% of students agreed that BlockPy's automatic guidance could be frustratingly vague. For example:

"The hints were not always helpful...they were very ambiguous sometimes which was frustrating."

"The most frustrating part of BlockPy was the tips that it gives you while creating an algorithm because if you were close, it didn't give you more hints as to where specifically the problem is, it just said you still have "a little way to go."

This reaction is partially biased by the timing of the survey. Earlier problems were intentionally equipped with more extensive suggestions and guidance than later problems, so students worked with less helpful problems at the time the students completed the survey. Regardless, it is clear that students wanted more guidance from the environment. Giving less guidance in later problems was partially motivated by a desire to have students work more on their own, but was also motivated by the time-intensive nature of developing more sophisticated guidance.

38% of the students suggested that error reporting in Spyder was more helpful than in BlockPy. In particular, they typically indicated that being able to identify the exact line that an error occurs in Spyder was tremendously helpful, and a major lack in BlockPy.

"It was able to tell me exactly what line of code was having an issue and what type of error."

Considering that the text mode of BlockPy does have this feature, we believe that students only considered the block interface. However, it is an understandable concern. Errors in BlockPy refer to line numbers instead of individual blocks, a disconnect that is recognized by the designers.

Table 3 reveals another concern: that students commit many errors that are currently undetected. The final submission from every student was analyzed using a flow-sensitive static-analysis algorithm that looked for code that, while valid and often matching the problem specifications, exhibited certain degenerative behavior. In the table, the first column is the type of semantic error, the second column is how many incidences were found in all student programs as a percentage of all programs submitted (50 students over 34 problems submitted 1587 programs), and the third column is how many incidences were found in programs marked correct as a percentage of all programs marked correct (1463 programs submitted that were correct). Figure 7 gives examples of some of these types of errors: declaring a variable that is never read, reusing the iteration list as the iterator, and in one case even iterating over an empty list. Often, these errors are silently unreported because they are guarded against by unreachable code paths, or have no impact on the program's output.

TABLE 3
Incidences of Semantic Errors Detected by Static Analysis of 1587 Student Programs and 1463 Correct Student Programs

	All	Correct
Changed type of variable	60.8%	62.1%
Variable overwritten without read	50.9%	51.6%
Variable never read	16.8%	14.3%
Variable read without write	12.0%	9.10%
Iteration list used in iteration	7.48%	6.29%
Iteration variable unused in iteration	6.53%	5.13%
Used iteration list as iteration variable	6.02%	5.88%
Iteration over non-list	1.58%	.75%
Used unknown function	.63%	.63%
Iteration over empty list	.06%	.07%

6 Future Work

We now outline future work and directions for BlockPy. Some of this work is technical, some is design decisions that must be revisited in light of evidence collected in its evaluation.

6.1 Improved Evaluation

Further investigation of the efficacy of the BlockPy environment and our curriculum is a top concern. Beyond the evaluation described above, further experimental studies are planned. Currently, we have a major, ongoing experimental study to better understand the impact of more dynamic feedback. Baseline data has been collected, in the form of both specially created pre/post student assessments and fine-grained log data. This data will inform future iterations of the environment, the curriculum, and publications. Beyond this experiment, we hope to evaluate other

components of BlockPy, including the dual block/text interface, and measure impact on student learning.

6.2 Improved Guidance

A major place for improvement is the automatic guidance system. Currently, the system requires much instructor effort, does not catch a number of problematic cases, and is not perceived as useful by its learners compared to other features. However, we believe that this feature has potential for helping students learn, based on the success of similar systems [40].

An addition to the environment now in progress is to integrate our static analyzer directly into the environment. A major outcome of this integration will be static type-checking of the block system, preventing a number of common, systematic student mistakes (e.g,. attempting to connect a scalar variable to the list slot of an iteration block). Although Python is a dynamically typed language, we believe that beginners can benefit from stricter type requirements.

6.3 The Data Science Process

BlockPy's stance on data science could be seen as "data science on rails". That is, there are specific datasets exposed through a preconceived interface. Often, students become most motivated when they are able to create their own datasets and their own problems. Although one solution is to broaden the number of datasets available, there is a long-term need for a general-purpose mechanism for users to access their own data sources through BlockPy. Previous work has been done to connect the Snap! programming environment with Google Sheets, so that students could access custom datasets [16]. Another major improvement to BlockPy would be to support the data science process at other phases, such as helping students to develop research questions or to interpret their visualizations.

7 Conclusion

We have described the design, development, and evaluation of a programming environment for beginners. It has a number of features including a dual text/block interface, a data science context, and guided feedback. Results from its use with introductory students suggest ways to improve the environment. We happily make our tool freely available at https://blockpy.com.

References

[1] J. Wing, "Computational thinking benefits society," *Social issues in computing*, 2014.

[2] M. R. Davis, "Computer coding lessons expanding for k-12 students," *Education Week*, 2013.

[3] K. Jordan, "Initial trends in enrolment and completion of massive open online courses," *The International Review of Research in Open and Distributed Learning*, vol. 15, no. 1, 2014.

[4] J. Wortham, "A surge in learning the language of the internet," *New York Times*, vol. 27, 2012.

[5] A. Forte and M. Guzdial, "Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses," *Education, IEEE Transactions on*, vol. 48, no. 2, pp. 248–253, 2005.

[6] P. K. Chilana, C. Alcock, S. Dembla, A. Ho, A. Hurst, B. Armstrong, and P. J. Guo, "Perceptions of non-cs majors in intro programming: The rise of the conversational programmer," in *Visual Languages and Human-Centric Computing, 2015 IEEE Symposium*, 2015, pp. 251–259.

[7] M. Goldweber, J. Barr, T. Clear, R. Davoli, S. Mann, E. Patitsas, and S. Portnoff, "A framework for enhancing the social good in computing education: a values approach," *ACM Inroads*, vol. 4, no. 1, pp. 58–79, 2013.

[8] H. Bort, M. Czarnik, and D. Brylow, "Introducing computing concepts to non-majors: A case study in gothic novels," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015, pp. 132–137.

[9] A. C. Bart, R. Whitcomb, D. Kafura, C. A. Shaffer, and E. Tilevich, "Computing with corgis: Diverse, real-world datasets for introductory computing," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017.

[10] P. Guo, "Python is now the most popular introductory teaching language at top us universities," *BLOG@ CACM, July*, 2014.

[11] R. Schutt and C. O'Neil, *Doing data science: Straight talk from the frontline*. " O'Reilly Media, Inc.", 2013.

[12] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15, 2015, pp. 91–99.

[13] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, 2015, pp. 101–110.

[14] ——, "To block or not to block, that is the question: students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*, 2015, pp. 199–208.

[15] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[16] J. D. Hellmann, "Datasnap: Enabling domain experts and introductory programmers to process big data in a block-based programming language," Master's thesis, Virginia Tech, 2015.

[17] B. Broll, A. Lédeczi, P. Volgyesi, J. Sallai, M. Maroti, A. Carrillo, S. L. Weeden-Wright, C. Vanags, J. D. Swartz, and M. Lu, "A visual programming environment for learning distributed programming," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*.

[18] A. Feng, E. Tilevich, and W.-c. Feng, "Block-based programming abstractions for explicit parallel computing," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*.

[19] Y. Ohshima, J. Mönig, and J. Maloney, "A module system for a general-purpose blocks language," in *Blocks and Beyond Workshop, 2015 IEEE*.

[20] D. Bau, M. Dawson, and A. Bau, "Using pencil code to bridge the gap between visual and text-based coding (abstract only)," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*.

[21] A. Altadmri and N. C. Brown, "Building on blocks: Getting started with frames in greenfoot 3," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*.

[22] S. H. Edwards, D. S. Tilden, and A. Allevato, "Pythy: Improving the introductory python programming experi-

ence," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 2014, pp. 641–646.
[23] T. Tang, S. Rixner, and J. Warren, "An environment for learning interactive programming," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*.
[24] P. 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*.
[25] S. Graham, "Skulpt," 2010.
[26] V. J. Shute, "Focus on formative feedback," *Review of educational research*, 2008.
[27] H. Keuning, J. Jeuring, and B. Heeren, "Towards a systematic review of automated feedback generation for programming exercises," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 2016.
[28] A. Gerdes, B. Heeren, J. Jeuring, and T. v. Binsbergen, "Ask-elle: an adaptable programming tutor for haskell giving automated feedback," 2015.
[29] B. D. Jones, "Motivating students to engage in learning: The MUSIC model of academic motivation," *International Journal of Teaching and Learning in Higher Education*, vol. 21, no. 2, pp. 272–285, 2009.
[30] J. Lave and E. Wenger, *Situated learning: Legitimate peripheral participation*. Cambridge university press, 1991.
[31] M. Guzdial, "Exploring hypotheses about media computation," in *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM, 2013, pp. 19–26.
[32] M. Guzdial and A. E. Tew, "Imagineering inauthentic legitimate peripheral participation: an instructional design approach for motivating computing education," in *Proceedings of the second international workshop on Computing education research*, 2006.
[33] D. Kafura, A. C. Bart, and B. Chowdhury, "Design and preliminary results from a computational thinking course," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 2015.
[34] R. E. Anderson, M. D. Ernst, R. Ordóñez, P. Pham, and S. A. Wolfman, "Introductory programming meets the real world: Using real problems and data in cs1," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 2014.
[35] K. Subramanian, J. Payton, D. Burlinson, and M. Mehedint, "Bringing real-world data and visualizations into data structures courses using bridges," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016.
[36] N. Fraser *et al.*, "Blockly: A visual programming editor," *URL: https://code. google. com/p/blockly*, 2013.
[37] M. Haverbeke, "Codemirror," 2011.
[38] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language migration in non-cs introductory programming through mutual language translation environment," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015.
[39] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper, "Mediated transfer: Alice 3 to java," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 2012.
[40] T. W. Price, Y. Dong, and D. Lipovac, "iSnap: Towards Intelligent Tutoring in Novice Programming Environments," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*.

Austin Cory Bart (Member, IEEE) received his PhD at Virginia Tech in Computer Science along with a certification in Learning Sciences. He received his undergraduate degree from the University of Delaware in Computer Science. His research interests are Computer Science Education, Software Engineering, and Program Analysis.

Javier Tibau (Member, IEEE) is a PhD candidate at Virginia Tech studying Computer Science. He is also an Assistant Professor at Escuela Superior Politecnica del Litoral (ESPOL). His research interests are in Human-Computer Interaction, and Computer Science Education.

Dennis Kafura (Member, IEEE) received his PhD from Purdue University. He is a Professor of Computer Science at Virginia Tech. He is the PI on two NSF IUSE awards both of which involved the development of a general education course in Computational Thinking at the university level using the BlockPy environment.

Clifford A. Shaffer (Senior Member, IEEE and Distinguished Member, ACM) received his PhD from the University of Maryland. He is Professor of Computer Science at Virginia Tech. His current research interests are in Computational Biology (specifically, user interfaces for specifying models and computations), Algorithm Visualization, and Computer Science Education.

Eli Tilevich (Senior Member, IEEE) received his PhD from Georgia Tech. He is an Associate Professor of Computer Science at Virginia Tech. His recent publications and current research focus on Software Engineering for Distributed and Mobile Computing and on Computer Science Education.