Design and Evaluation of an Introductory Computing Environment

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 limited goals. We seek to improve curriculum 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 available [at https://www](http://www.blockpy.com/).blockpy[.com.](http://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 automatic guidance to learners as they complete problems, but also mediates transfer to more sophisticated programming environments by supporting bidirectional, seamless transitions between block and text interfaces. 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, a two-week 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, Intelligent Tutor, Automatic Guidance.

✦

1. **INTRODUCTION**

As computing has become pervasive across ca- reers 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 specialized educational approaches, as compared 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

*Computer Science, College of Engineering, Virginia Tech, Blacks- burg, 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.*

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. In this paper, we describe the design of BlockPy, evaluate its use in a real-world classroom environment, and describe future directions for BlockPy in particular and for introductory computing environments in general. The intended audience of the paper is educators interested in using BlockPy and developers who would wish to develop systems similar to BlockPy. This paper makes a number of design contributions.

* 1. A scaffolded, web-based environment for novice programmers,
  2. Integration of authentic, real-world data sets into the environment, and
  3. A feedback authoring API and an abstract interpreter for static analysis

The paper makes further contributions to evaluate the system, with the following results:

1. Most, but not all, of the features in the scaf- folded learning environment are effective,
2. Novice learners can successfully solve data science problems within the environment, and
3. Improved feedback is needed
4. **THE *Why* OF BLOCKPY**

BlockPy draws inspiration from a number of theoreti- cal and concrete sources. Design decisions were influ- enced by educational theories, existing introductory programming environments, and the wider commu- nity of professional software developers.

# Python and Blocks

BlockPy, as its name suggests, is a block-based editor for the Python programming language. Python is a popu- lar introductory languages because of its beginner- friendly syntax, powerful library, and popularity among professional programmers [9]. So not only are learners likely to be successful when working with Python, but they can quickly enter into an authentic community of practice and start solving real-world problems. Python also has a well-earned reputation as a useful language for performing Data Science [10], en- suring 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, decreas- ing their start-up time and helping them accomplish tasks they originally could not [11], [12]. Blocks help beginners navigate their program’s structure while preventing syntax errors. They can also visually and clearly expose a complex API, such as those used in game development or data processing. These benefits offset the major disadvantage of blocks: learners can negatively perceive block interfaces as being only for younger learners or unsuitable for professionals [13].

BlockPy has much in common with other block-based programming environments. For example, Scratch and its successor Snap! [14] largely target young learners, both in design and with their game development context. Extensions to Snap! have integrated more sophisti- cated program features, although these have been lim- ited. Hellman [15] incorporated Data Science features, including access to real-time datasets, user-created data sources, and cloud-based data manipulation. The NetsBlox project [16] exposes distributed computing concepts by introducing event blocks for network transmissions. Others have promoted patterns for par- allel programming abstractions within Snap!, such as producer-consumer and MapReduce [17].

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. De- veloped 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 inter- face. 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” [18]. A potential criticism of this approach to authenticity is that, instead of a using an existing popular language, they use a language descended from Squeak.

BlockPy has much in common with other block-based editors for mainstream lan- guages. PencilCode is a JavaScript editor that offers a seamless transition between blocks and text [19]. GreenFoot is a visual programming environment for creating games and animations in Java, with an inno- vative structured code editing interface they refer to as “Frames” [20]. There is a clear trend in modern editors to support a dual-interface between both blocks and text in order to transition students gracefully.

BlockPy is not the first web-based Python execu- tion environment, but it advances the state-of-the-art established by its predecessors, including Pythy [21], CodeSkulptor [22], and the Online Python Tutor [23]. None of these systems support a dual block/text interface. Both CodeSkulptor and Pythy are built on the same underlying engine, Skulpt [24], which can cross-compile Python code to JavaScript. CodeSkulp- tor has an extensive but custom API for creating user interfaces and games, which is powerful but limits stu- dents’ ability to transfer code away from the browser. CodeSkulptor is intended as an undirected environ- ment for creativity, but therefore does not guide learn- ers through a curriculum. Pythy, on the other hand, is an assignment-oriented application with limited sup- port 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 uses remote code exe- cution to provide visualizations of users’ algorithms. Although rigorous and detailed, the OPT is not ideal for learners who must parse the complicated terminol- ogy being used. Further, the Online Python Tutor is an undirected environment like CodeSkulptor, rather than a platform for a curriculum. Finally, its depen- dency on a remote server makes the platform vulner- able to poor internet connections and complicates the applications’ architecture.

A final point of comparison is BlockPy’s guided feedback. Surprisingly few introductory environments give interactive and guiding feedback to students as they run their code, usually at best relying on unit tests. CodeCademy is perhaps the most popular and successful guided feedback platform. Unfortunately, CodeCademy is closed-source and has not published information about the efficacy of its curriculum or its techniques.

# 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 . This ing, Appealing to a student's sense of usefulness or interest strengthens their The second theory is [26].This

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 artwork and music. Al- though there is established motivational value in this approach [27], 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 [28], 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 it is unclear how 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 very 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 Sci- ence” 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 au- thoritative 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.

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 comput- ing curriculum and pedagogical decisions surround- ing 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 manip- ulating data. BlockPy is designed as an active learning environment, with an emphasis on students interact- ing 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.

The authors have had prior success in using a Data Science context through the CORGIS project, which makes it easier to introduce real-world datasets into an introductory computing curriculum in order to motivate students. This project has been deployed in a college-level Computational Thinking course, with great effect on student motivation [29], [30]. This paper does not further evaluate the data science context, but focuses instead on the scaffolds developed for the environment.

1. ***BlockPy* ’S MAJOR FEATURES**

In this section, we briefly describe the major features of BlockPy. 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 is visualized. The goal of BlockPy is to make itself unnecessary, and graduate the learner into a professional programming environment. Scaffolding eases this transition and, where possible, the environ- ment attempts to maintain an authentic programming experience. Figure 1 shows the version of BlockPy

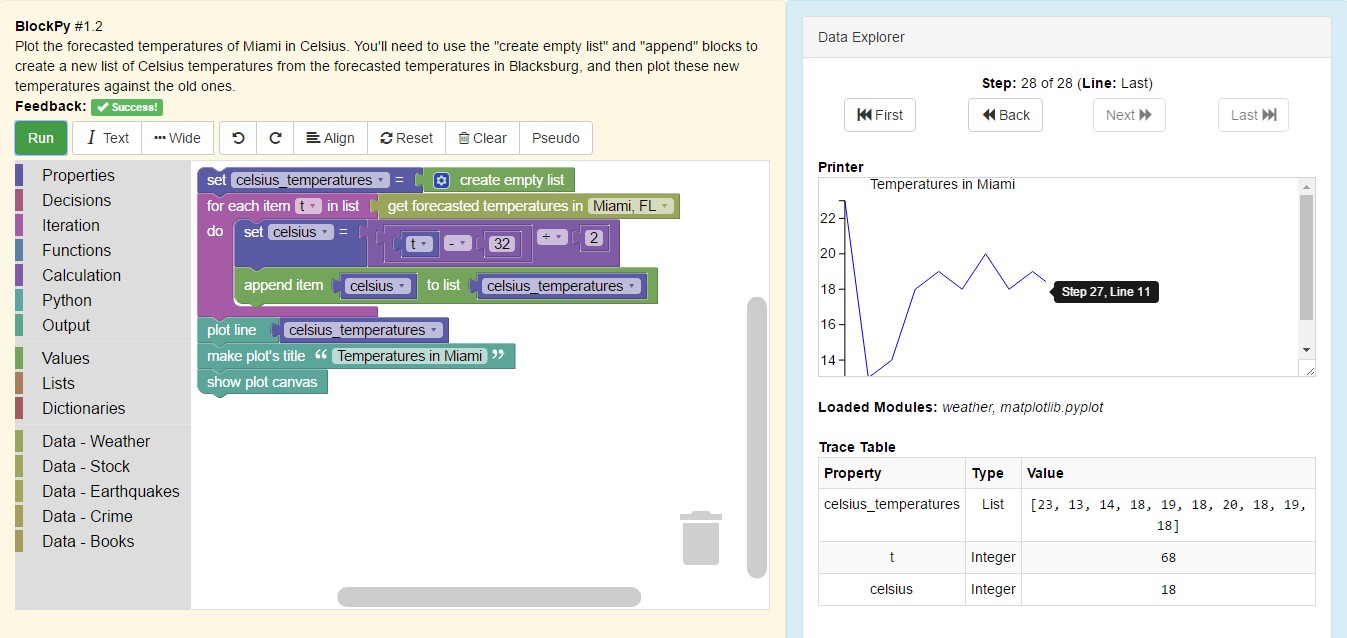


Fig. 1. A Screenshot of BlockPy in Action

used in the associated study, while recent versions changes some of the layout.

**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 Fig- ure 2. The block interface uses the Google Blockly library [33], while the text interface uses the CodeMir- ror library [34]. 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** Although the block interface is a useful mechanism for introductory programmers to avoid syntax errors, the most valuable pedagogical component of BlockPy is its Guided Feedback system. Through a custom interface, instructors can define a function that takes the student’s code, a trace of its execution, and any output; the function can then re- turn either HTML feedback or an indication that they have successfully completed the problem. BlockPy is therefore able to give guidance that is closely tuned to the problem. BlockPy’s regular error messages have been extended with additional information for be- ginners. Guided feedback is particularly aligned with BlockPy’s emphasis on active learning, since feedback is one of the most crucial elements of learning.

**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, with no round trip to a server. Students can continue to work even without an internet connection. Complicated sandboxes are not necessary 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 unexposed (except through explicit APIs, discussed in a later section).

**Natural Language Code Description** Another scaf- fold of BlockPy is a natural language program descrip- tion 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 gen- erate a string of conventional English text. It can be seen as a third phase to the existing dual text/block 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 introduc- tory students often struggle to trace the execution of their programs on their own. Using the property explorer, not only can students observe the appear-

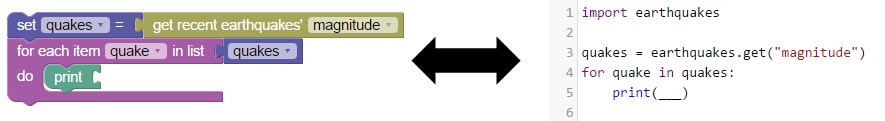


Fig. 2. Mutual Language Translation between Block and Text interfaces

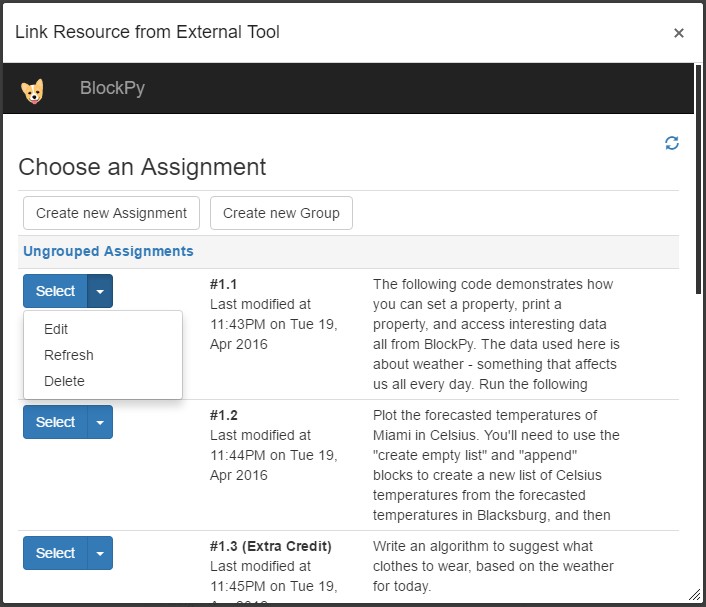


Fig. 3. Course Assignment Management

ance 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. To achieve this, BlockPy natively integrates CORGIS, the Collection of Real- time, Giant, Interesting dataSets. CORGIS contains diverse data, from authoritative sources. The subset of CORGIS exposed in the current BlockPy interface is selected based on perceived popular appeal, sim- plicity, and pedagogical affordances of the data. These libraries are available through simple blocks. These blocks translate into function calls that return struc- tured 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 demon- strate these blocks in action.

**Plotting** Another addition to Skulpt is tools for making visualizations, a key mechanism of Data Sci- ence. Currently, BlockPy supports the creation of line plots and scatter plots. We rely largely on prior work to support the creation of statistical visualizations within Skulpt, using an API identical to the popular MatPlotLib API [35]. By mimicking this professional API, BlockPy increases its authenticity and promotes deeper transfer.

**LTI Support** Although BlockPy has its own in- ternal course management system, it also supports

LTI (Learning Technology Interoperability). This im- portant standard separates “Tool Consumers” from “Tool Providers”. 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 com- plete problems in the environment, they are automat- ically graded, and the LMS is given the submission.

One service provided by LTI is that users are automatically added into the BlockPy database, au- thenticated through Canvas. When a student loads a BlockPy problem, Canvas delivers their email 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. To support this educational technology standard, we have started work on spinning off a template for future Python-based LTI applications. This open- source project is available at https://github.com/

acbart/lti-flask-skeleton.

1. **DESIGN ISSUES**

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

# Internal Code Representation

Dual block/text editors create an added challenge in deciding the right internal representation of the stu- dents’ code. BlockPy uses the textual representation as the canonical representation, as opposed to the block parse tree. 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 prop- erties to individual lines such as geometric informa- tion [19]. A major limitation of both representations is that some valuable programmer-level semantic data is not preserved. On the text side, user-created whites- pace 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 [36] solved this problem by storing geo- metric information of blocks in the comments of the text mode. However, this leads to crowded code with

import stocks

stocks = stocks. get past("FB") new = []

for stock in new: new = []

for stocks in stocks:

print(stocks)

Fig. 4. Degenerate Student Code

confusing comments. The trade-offs in this system led to BlockPy’s design as a primarily text-driven environment under the hood.

# 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 more explicit plain text phrasing, to explain the nature of the block more clearly: for each item [ ] in list [ ]. Similarly, the as- signment 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.

# 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 stu- dents to write invalid code from the text editor, making the transition back to blocks problematic. A missing colon, unclosed parentheses, or incorrect in- dentation will prevent Skulpt from generating a valid parse tree. When encountering code with a syntax error, BlockPy creates “raw blocks” that store the lit- eral 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 con- founded into creating one large raw block. Although some might consider this a major disadvantage, it is not necessarily desirable to ensure that students are always writing completely valid programs at all times, especially in the early stages of constructing an algorithm, particularly when new programming con- structs are introduced. Although BlockPy is built on the premise that it is worth delaying the conversation about syntax, all students must eventually become

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 |

comfortable with the details of writing structurally correct code.

It is not always possible to automatically correct a students’ written code to match their intent (par- tially because some students may not even understand their intent!). However, there are more sophisticated approaches to improving the support given to stu- dents. 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. Rivers and Koedinger explore other approaches that uses prior student submissions for each problem to suggest corrections to the user [37].

1. **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 ma- jors 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.

In the Spring 2016 offering, 50 students were as- signed 34 BlockPy problems over the course of 4 classes over 2 weeks (see Table 1). This focused use of block programming builds upon their notions of algorithm,s which they had previously seen during a 6-week introductory unit where students created algo- rithms 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 (usually in the form of a Parson’ problem). Homework questions would have them complete similar problems from scratch. After the BlockPy unit, the curriculum continued in the Spyder IDE, as a way to transition students into a more authentic setting. To facilitate this shift, the last day of BlockPy material started in text mode, and

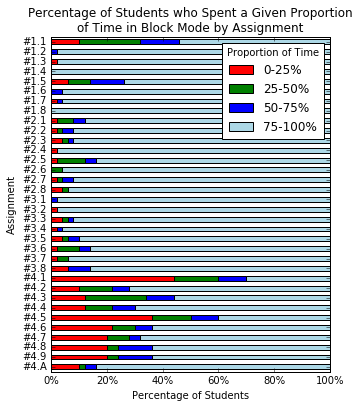
IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING 7

TABLE 2

BlockPy Survey Qualitative Results

|  |  |  |  |
| --- | --- | --- | --- |
| Environment | State | Tag | Percentage |
| BlockPy | Helpful | Block Interface | 56.3% |
|  |  | Dual Text/Block | 29.2% |
|  |  | Guidance | 10.4% |
| BlockPy | Frustrating | Vague Guidance | 31.2% |
| Spyder | Helpful | Better errors | 31.2% |
|  |  | Variable Explorer | 31.2% |
|  |  | Writing text code | 8.3% |

encouraged students to become familiar with writing code in that form. The BlockPy curriculum was not intended to be the remainder of the curriculum, and was faded in the transition to Python. In contrast to other curricula [38], for our college-level learners the BlockPy scaffolding could be removed after 4 days. More information about the curriculum can be found in [29], and at the public site for the course [(http://think.cs.vt.edu/course](http://think.cs.vt.edu/course) materials/).

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 two weeks after the BlockPy component was com- pleted. 41 students gave consent for their answers to be released for research purposes, with 19 male students, and 22 female. 32% were freshmen, 39% were sophomores, 17% were juniors, and 12% were seniors.

The survey asked a multiple-choice question about the where students felt that the BlockPy/Spyder tran- sition should occur. “When do you think we should have STOPPED using BlockPy and started using Spy- der?” In addition to an “Other” response, six alter- natives, including an “Always” and a “Never”, “The Same”, and 3 points in the course, implying more or less use of BlockPy. The results of this question are discussed in Section 5.1.

This survey also had four free-response questions asking students about frustrating and helpful fea- tures in both BlockPy and Spyder. These free-response questions were qualitatively coded using a grounded- theory approach, resulting in generalized tags of stu- dent responses. Sections 5.1 and 5.4 describe the re- sults of the analysis of this part of the survey.

# The Environments’ Effectiveness

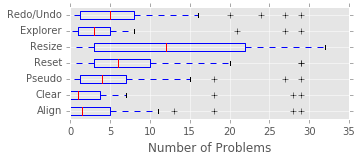
In the free response section, students were positive about BlockPy’s block interface. 65% indicated the block interface as particularly helpful, and 34% indi- cated that the dual text/block interface helped. Con- sistent with the criticism of the automatic guidance,

Fig. 5. Percentage of Students who Spent a Given Proportion of Time in Block Mode by Assignment

only 12% found that feature particularly helpful; our interpretation from this result is that students ap- preciated the guidance, but wanted more from it. In terms of other frustrations, 7 students (17%) suggested issues with using blocks (e.g., having to repeatedly drag blocks around). Other frustrations were with the problems assigned or the nature of coding in general. Comparatively, most criticisms of Spyder related to coding in general, rather than features of the envi- ronment (e.g., students described frustrations with Python syntax, trying to interpret errors).

Figure 5 shows the percentage of students who spent different proportions of time in block mode, over assignments. In other words, the red bars rep- resent students who spent very little time in the Block mode (and were therefore predominately writing text), while the light-blue bars represent students who were predominately using the Block mode. 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 should be given to direct students to pay attention or take advantage of the text interface, to build their competency with that form of their programs.

The last day of BlockPy began the transition to

Fig. 6. Students’ Use of Toolbar Features over Problems

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 and Python text.

One effect observed in the transition to text was that students were confused what the keywords of Python were, compared to BlockPy. As previously described, the 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 tran- sition 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: 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 cor- rected 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. Nonetheless, it does seem prudent for the guidance to correct students who might be using such incorrect code forms.

The main toolbar in BlockPy gives students ac- cess to a number of features intended to help them complete problems. Some of these tools are standard editor features, such as undo and redo buttons. Some of them, however, are more specific to the educa- tional nature of the environment, such as the code

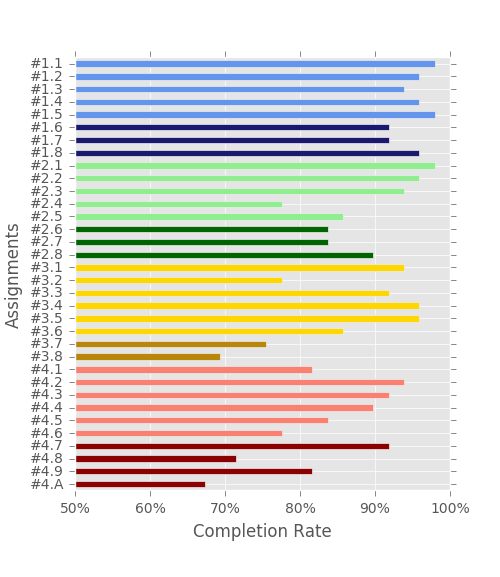
explorer and natural language code generator (named pseudo in this version). Figure 6 reports on student use of these features across problems. Unfortunately, these data suggest that most students did not take advantage of these features. Deeper analysis found no significant correlations between student performance (as established by either time on task or number of successfully completed problems) and level of use of the toolbar features. The x-axis is the number of problems where a student used the particular feature at least once. In part, we attribute this underutilization to these features not being sufficiently emphasized in the instruction.

# The Context Manageability

The Data Science context embedded in the curricu- lum was manageable by the students, based on the completion rates and student survey responses. Fig- ure 7 describes student completion rates over the assignments. The bars are colored based on day and their brightness indicates homework vs. classwork. As can be seen, there are 4 days of BlockPy activities. Figure 8 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, which sug- gests that most students were able to complete most problems. The average student was able to complete most problems in 15 minutes, which was considered reasonable by the instructors.

Although the aggregate set of problems was promising, inadequacies exist within the curriculum. Some problems seemed to take students much longer than anticipated, such as #3.7, which had students write code from scratch to identify the index of the minimum value in a list. As students progressed through the curriculum, there is a visible reduction in the percentage of completions. There is a similar reduction from classwork to homework. While the completion rate almost always remains above 70%, there was some noticeable drop-off on the last day. This is most likely caused by three things: fatigue on the part of the students (accumulated work load might have been overwhelming), the proximity to the end of the unit (which means that students had comparatively less time to work on these problems than earlier problems), and the fact that day 4 is the transition to text mode and students struggle with the new syntax.

The survey asked students where they would pre- fer to make the transition from BlockPy to Spyder, and allowed to choose from a series of intervals in the course. The majority (28 student) felt that the current location was appropriate, 6 students felt it should be later, 6 felt instruction should be kept in BlockPy,

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% |

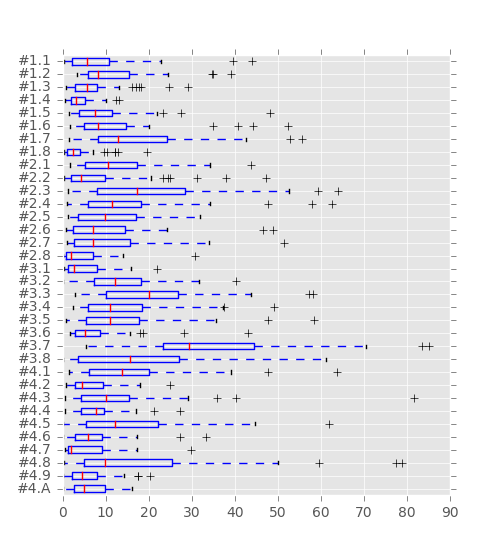
Fig. 7. Completion Rate by Problem (colors indicate distinct days, brightness indicates homework vs classwork)

Fig. 8. Work Session Length (minutes) by Problem

and 7 students felt 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 more mature environment, and so is not meant to last for too much of the course.

# The Feedback Quality

In the free response section, there was little agreement about what was most frustrating about BlockPy, with one major exception: 34% of students agreed that BlockPy’s automatic guidance could be frustratingly vague. This is partially biased by the timing of the survey. Earlier problems were (somewhat) intention- ally equipped with more extensive suggestions and guidance than later problems, so students were most recently working with less helpful problems. Regard- less, it is clear that students reacted negatively to the reduction in guidance. 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.

Students suggested that error reporting in Spyder was more helpful than in BlockPy. In particular, they indicated that being able to identify the exact line that an error occurs in Spyder was tremendously helpful and a major lack in BlockPy. Considering that the text mode of BlockPy does have this feature, we believe that students were only considering the block inter- face when making this assessment. However, it is an understandable concern. Errors in BlockPy refer to line numbers instead of individual blocks, a disconnect that was not considered enough by the designers.

Table 3 reveals a bigger concern than lack of guidance or inexact error reporting. The final sub- mission 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 4 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 code.

1. **FUTURE WORK**

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

# Improved Evaluation

Further investigation of the efficacy of the BlockPy environment and our curriculum is a top concern for the project moving forward. Although the evaluation described above was a first step in evaluating the sys- tem, experimental studies are necessary to determine the optimal development of features. 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 their impact on student learning and success.

# Improved Guidance

A major place for improvement in BlockPy is the auto- matic guidance system. Currently, the system requires too much instructor effort, does not catch a number of problematic cases, and is not perceived to be as useful by its learners compared to other features. However, we believe that this feature has the most potential for helping students learn.

A hurdle for instructors is the cumbersome nature of authoring guidance. We are designing a new inter- face to streamline types of feedback that instructors most often give. Some of these features are related to ensuring that the students’ output matches ex- pectations. For example, symbolic program analysis can be used to ensure that students’ output matches certain general formula instead of specific strings. Other features are designed to let instructors enforce restrictions about students code: that they use certain language constructs, or that they have a declaration for a particular type of variable. We are guided by

recent work by Singh et al, and Rivers et al. [37], [39], although we believe that an increased emphasis is required on the role of the instructor to provide particular pedagogical details for assignments.

An addition to the environment now in progress is to integrate our static analyzer directly into the environment, to improve feedback. A major outcome of this integration will be static type-checking of the block system, preventing a number of common, sys- tematic student mistakes (e.g,. attempting to connect a scalar variable to the list plug of an iteration block). Although Python is a dynamically typed language, we believe that beginners can benefit from stricter type requirements. Beyond type checking, we need to provide more support for students to understand syn- tax and run-time errors, particularly error messages reported by the Python itself.

We are planning improvements to interventions made after the environment has detected errors by the learner. Currently, the only feedback delivered by the environment are error reports, instructor written HTML snippets, and reporting of successfully com- pleted problems. We envision a much richer system. First, learning resources should be made directly avail- able as needed; for example, relevant chapters of the open-access course book or short instructional ani- mations and videos. Second, the environment should prompt the learner to take advantage of its pedagog- ical tools; students may not be using features like the property explorer and natural language code expla- nation because they may not have the metacognitive ability to know how it would benefit a given problem. Third, the environment can encourage learners’ peers and instructors to intervene in a situation, or at least notify the course staff if a student is struggling with a particular concept or for a long time.

# Tiered Block Interface

Transitioning students from the block interface to the text interface and eventually to a professional envi- ronment remains an unsolved problem. Although stu- dents seem to handle this fairly well, they did suggest some difficulties in the survey. We believe that estab- lishing a more gentle gradient between blocks and text can assist in the transition. We propose using a tiered block interface to gradually shift from more verbose blocks into blocks that mimic literal Python syntax more closely (e.g., for each ... would change to for). At some point in the curriculum, the interface would change to the less verbose blocks in preparation for the eventual change in modes. This discrete change could also be supported graphically by the blocks moving closer and closer to regular text. For example, PencilCode uses faded transitions to suggest a con- tinuous transition between blocks and text [19], and

Greenfoot 3 uses a purposefully structured interface to make blocks seamlessly mimic text [20]

# Missing Language and API Features

Although the underlying Skulpt execution environ- ment is a full Python implementation, the entire li- brary is not supported (including internal libraries such as SQL and popular third party libraries such as Pandas or SciPy). Additionally, the block interface does not have bindings for every syntactical language element. Notable missing elements include try/except blocks, with blocks, lambda expressions, and inline list comprehensions. Finally, the CORGIS library has a large number of other APIs that are not currently available through the interface. Development of the environment is driven by the needs of our curriculum. Although this is partially born of practicality, there is a tactical value to letting the interface emerge organi- cally.

# Missing Contexts

BlockPy was built around a Data Science context, with the hypothesis that this would be almost-universally appealing for students. However, some disciplines and age groups may find Data Science to be uninter- esting. Other approaches to introductory computing have their own motivational and pedagogical benefits. For instance, animation and game design have both

explore 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 envi- ronment with Google Sheets, so that students could access custom datasets [15]. Another major improve- ment 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 visu- alizations. Students could be tasked with completing other phases of the Data Science process formally from within the environment.

1. **CONCLUSION**

We have described the design, development, and eval- uation of a programming environment for beginners. It has a number of features including a dual text/block interface, a data science context, and immediate feed- back. Results from an intervention with introductory students suggests ways to improve the environment. We happily make our tool freely available.

**REFERENCES**

[1] J. Wing, “Computational thinking benefits society,” *Social*

proven to be valuable contexts, albeit with a different design philosophy. Although preliminary results gath-

[2]

*issues in computing*, 2014.

M. R. Davis, “Computer coding lessons expanding for k-12 students,” *Education Week*, 2013.

ered in our research suggest the appeal of Data Science

over other contexts, our results are not conclusive,

[3] K. Jordan, “Initial trends in enrolment and completion of massive open online courses,” *The International Review of*

and there is not a clear disadvantage to most other contexts.

[4]

*Research in Open and Distributed Learning*, vol. 15, no. 1, 2014.

J. Wortham, “A surge in learning the language of the internet,” *New York Times*, vol. 27, 2012.

In terms of pedagogical benefits, block-based pro-

gramming environments can help make abstract data more concrete. The official Blockly project provides Blockly Games, including an activity where users direct an avatar through a maze using simple Tur- tle Graphics-like commands (e.g., “move”, “turn”). BlockPy has limited support for the Maze activity, but only so that it can be incorporated as an LTI assign- ment. It shares no client-side code with the primary BlockPy interface, and is not meant to be a part of the official environment. However, there are no technical reasons why BlockPy could not be extended to work for other contexts and to support other paradigms of introductory programming. A certain amount of development is required, though.

# 6.6 The Data Science Process

BlockPy’s take on Data Science could be seen as “Data Science on rails”. That is, there are specific datasets exposed through a preconceived interface. Often, stu- dents become most motivated when they are able to

1. A. Forte and M. Guzdial, “Motivation and nonmajors in computer science: identifying discrete audiences for intro- ductory courses,” *Education, IEEE Transactions on*, vol. 48, no. 2, pp. 248–253, 2005.
2. P. K. Chilana, C. Alcock, S. Dembla, A. Ho, A. Hurst,

B. Armstrong, and P. J. Guo, “Perceptions of non-cs ma- jors in intro programming: The rise of the conversational programmer,” in *Visual Languages and Human-Centric Com- puting, 2015 IEEE Symposium*, 2015, pp. 251–259.

1. M. Goldweber, J. Barr, T. Clear, R. Davoli, S. Mann, E. Patit- sas, 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.
2. H. Bort, M. Czarnik, and D. Brylow, “Introducing com- puting 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.
3. P. Guo, “Python is now the most popular introductory teaching language at top us universities,” *BLOG@ CACM, July*, 2014.
4. R. Schutt and C. O’Neil, *Doing data science: Straight talk from the frontline*. ” O’Reilly Media, Inc.”, 2013.
5. T. W. Price and T. Barnes, “Comparing textual and block interfaces in a novice programming environment,” in *Pro- ceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER ’15, 2015, pp. 91–99.
6. D. Weintrop and U. Wilensky, “Using commutative as- sessments 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.
7. ——, “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.
8. M. Resnick, J. Maloney, A. Monroy-Herna´ndez, N. Rusk,

E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Sil- ver, B. Silverman *et al.*, “Scratch: programming for all,” *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

1. 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.
2. B. Broll, P. Vo¨ lgyesi, J. Sallai, and A. Le´deczi, “Netsblox: a visual language and web-based environment for teaching distributed programming,” 2016.
3. A. Feng, E. Tilevich, and W.-c. Feng, “Block-based program- ming abstractions for explicit parallel computing,” in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 71–75.
4. Y. Ohshima, J. Mo¨ nig, and J. Maloney, “A module system for a general-purpose blocks language,” in *Blocks and Be- yond Workshop, 2015 IEEE*, Oct 2015, pp. 39–44.
5. 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*, ser. SIGCSE ’15, 2015, pp. 706–706.
6. 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*. ACM, 2016, pp. 715–715.
7. 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.
8. 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*, ser. SIGCSE ’14, 2014, pp. 671–676.
9. 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 Educa- tion*, ser. SIGCSE ’13, 2013, pp. 579–584.
10. S. Graham, “Skulpt,” 2010.
11. 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.
12. J. Lave and E. Wenger, *Situated learning: Legitimate peripheral participation*. Cambridge university press, 1991.
13. M. Guzdial, “Exploring hypotheses about media com- putation,” in *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM, 2013, pp. 19–26.
14. M. Guzdial and A. E. Tew, “Imagineering inauthentic legit- imate peripheral participation: an instructional design ap- proach for motivating computing education,” in *Proceedings of the second international workshop on Computing education research*. ACM, 2006, pp. 51–58.
15. 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*. ACM, 2015.
16. 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*, New York, NY, USA, 2017, pp. 57–62.

1. R. E. Anderson, M. D. Ernst, R. Ordo´ n˜ ez, 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*, ser. SIGCSE ’14, 2014, pp. 465–466.

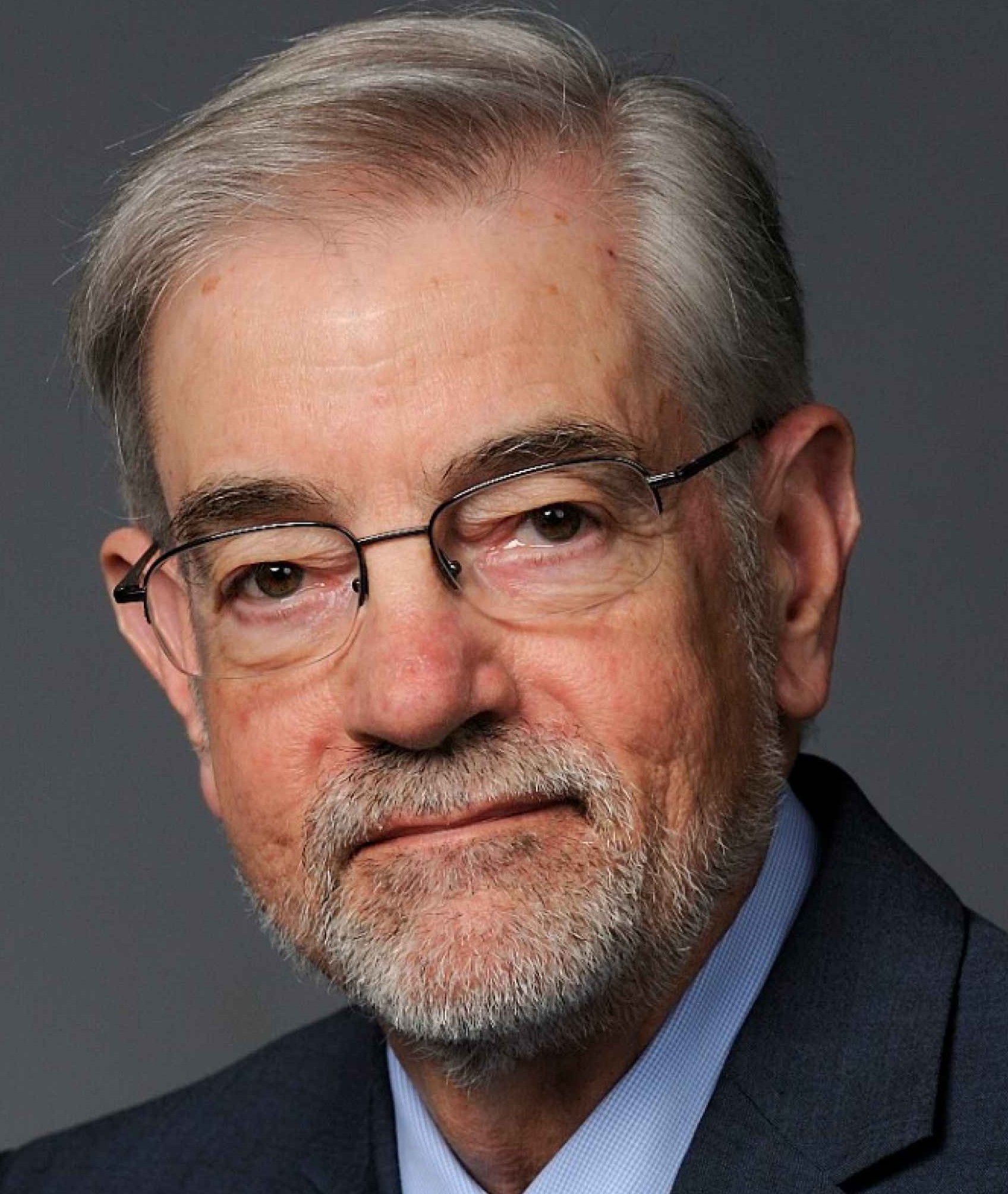
1. K. Subramanian, J. Payton, D. Burlinson, and M. Mehedint, “Bringing real-world data and visualizations into data structures courses using bridges,” in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 590–590.
2. N. Fraser *et al.*, “Blockly: A visual programming editor,”

*URL: https://code. google. com/p/blockly*, 2013.

1. M. Haverbeke, “Codemirror,” 2011.
2. M. Ebert, “Skulpt matplotlib,” https://github.com/ waywaaard/skulpt matplotlib, 2014.
3. Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, “Lan- guage migration in non-cs introductory programming through mutual language translation environment,” in *Pro- ceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’15, 2015, pp. 185–190.
4. K. Rivers and K. R. Koedinger, “Data-driven hint genera- tion in vast solution spaces: a self-improving python pro- gramming tutor,” *International Journal of Artificial Intelligence in Education*, pp. 1–28, 2015.
5. 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 Educa- tion*, ser. SIGCSE ’12. New York, NY, USA: ACM, 2012, pp. 141–146.
6. R. Singh, S. Gulwani, and A. Solar-Lezama, “Automated feedback generation for introductory programming assign- ments,” in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 15–26.

**Austin Cory Bart** 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 Com- puter Science. His research interests are Computer Science Education, Software Engineering, and Program Analysis.

**Javier Tibau** is a PhD candidate at Vir- ginia 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 Com- puter Science Education.

**Dennis Kafura** 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 gen- eral 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 re- search interests are in Computational Biology (specifically, user interfaces for specifying models and computations), Al- gorithm Visualization, and Computer Sci- ence Education.

**Eli Tilevich** (Senior Member, IEEE) re- ceived his PhD from Georgia Tech. He is an Associate Professor of Computer Science at Virginia Tech. His recent pub- lications and current research focus on Software Engineering for Distributed and Mobile Computing and on Computer Sci- ence Education.