

# Design and Evaluation of an Open-access 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 <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 careers and disciplines, students and professionals alike are expected to develop computational skills and thought processes [1]. Efforts to address these 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 completely individualized, informal learning platforms (e.g., CodeCademy) [4]. Students in these courses have dissimilar motivations, clarity of goals, and depth of prior experiences [5], [6]. Consequently, this growing population need support from specialized educational approaches, as compared to traditional Computer Science students [7], [8].

We seek to overcome these problems by means of BlockPy, a web-based, introductory programming en-

vironment. An open-source project, BlockPy is available at <https://www.blockpy.com>. The major design philosophies of the environment emphasize Data Science as an authentic learning context and scaffolds students with a friendly dual text/block languages and guided instruction. In this article, we describe the design and development 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.

## 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 Educational Theories

BlockPy conforms to motivational and cognitive educational theories of learning, and principles of instructional design.

A major issue for introductory students is motivation, not only to complete individual problems

- Computer Science, College of Engineering, Virginia Tech, Blacksburg, VA, USA.

Corresponding Author: Austin Cory Bart ([acbart@vt.edu](mailto: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.

and assignments, but to engage with computing curriculum and understand its role in their long-term career and life goals [5]. Motivation is a complex, multi-faceted construct with many interpretations. **The MUSIC Model of Academic Motivation** [9] is a meta-descriptive theory, which aggregates other motivational theories into five major components of learner motivation: Empowerment, a sense of personal agency; Usefulness, a sense that the material is connected to long-term goals; Success, a sense that the learner can achieve the goals; Interest, a sense of intrigue and inspiration; and Caring, a sense of socialness within the experience. One or more components must be present for a student to become intrinsically motivated [10].

Many introductory computing contexts focus on fostering a sense of interest. Media Computation, for instance, is a creativity-based curriculum by Guzdial et al. where students create artwork and music. Although there is established motivational value in this approach [11], 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 [12]. Situated Learning Theory describes how learners become particularly engaged with material when they can perceive its **authenticity** and connection to a Community of Practice that they might reasonably become a part of. When choosing a foundational context for Blockly, establishing a sense of authenticity seemed crucial.

Blockly is designed as an active learning environment, with an emphasis on students interacting and receiving feedback. This means that Blockly is designed to require minimal amounts of instruction and presentation of material. **At the same time, a major theme of Blockly 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.

## 2.2 Data Science as an Introductory Context

Blockly is part of a growing movement within Computer Science Education to promote “Data Science” as an authentic context for 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. This is partially in response to the rise of curriculums that focus on games and animation design. These represent “interest-based” curricula instead of “usefulness-based” curricula.

Blockly is built around the Data Science context, and provides tools for students to rapidly begin working with real-world datasets relevant to their personal

and work interests. The computing curriculum and pedagogical decisions are heavily influenced by this context. Lessons are built around collection-based iteration, for instance, as opposed to conditional iteration. Tools are also provided by the environment for conveniently visualizing and manipulating data.

The authors have had prior success in using a Data Science context through the CORGIS project [13], which makes it easier to introduce real-world datasets into an introductory computing curriculum. This project has been successfully deployed in a college-level Computational Thinking course [14], described in more detail later in this paper.

Other research initiatives have promoted Data Science in introductory computing. Anderson et al. suggested a curriculum built around real-world projects and data sources [15]. Goldweber et al. [7] developed an entire framework for evaluating and designing projects with this theme. The BRIDGES project [16] 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.

## 2.3 Python and Blocks

Blockly, as its name suggests, is an editor for the Python programming language. Python is a popular introductory language because of its beginner-friendly syntax, powerful library, and popularity among professional programmers [17]. **So not only are learners more 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 [18], ensuring a harmonious relationship between Blockly’s language and context.

Block-based languages have proven themselves as a powerful scaffold for novice learners, decreasing their start-up time and helping them accomplish tasks they originally could not [19], [20]. 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 childish or unsuitable for professionals.**

Scratch and its successor Snap! are well-known block-based programming environments [21]. They largely target young learners, both in design and with their game development context, contributing to the popular image of block languages as toys for kids. Extensions to Snap! have integrated more sophisticated program features. Hellman [22] incorporated

Data Science features, including access to real-time datasets, user-created data sources, and cloud-based data manipulation. The NetsBlox project [23] 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 [24]. However, all of these extensions are still embedded in the Scratch/Snap! language, which bears little-to-no relation to the languages seen in introductory courses (i.e. Python, Java, Racket).

GP, the “The Extensible Portable General Purpose Block Language for Casual Programmers”, 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” [25]. A potential criticism of this approach to authenticity is that, instead of using an existing popular language, they use a language descended from Squeak.

Modern block-based editors attempt to bring the benefits of block interfaces to more mainstream languages. PencilCode is a JavaScript editor that offers a seamless transition between blocks and text [26]. 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” [27]. Although both of these editors gracefully bridge text and blocks, they are tied to contexts that are, arguably, not authentic. Still, 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 execution environment, but it advances the state-of-the-art established by its predecessors, including Pythy [28], CodeSkulptor [29], and the Online Python Tutor [30]. None of these systems support a dual block/text interface. Both CodeSkulptor and Pythy are built on the same underlying engine, Skulpt [31], 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 uses remote code execution to provide visualizations of users’ algorithms. Although rigorous and detailed, the OPT is not ideal for learners who must parse the complicated terminology being used. Further, the Online Python Tutor 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. Surprisingly few introductory environments are designed to 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 platform. Unfortunately, CodeCademy is closed-source and has not published information about the efficacy of its curriculum or its techniques.

### 3 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 environment attempts to maintain an authentic programming experience. Figure 1 shows the version of BlockPy 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 Figure 2. The block interface uses the Google Blockly library [32], while the text interface uses the CodeMirror library [33]. 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 return 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

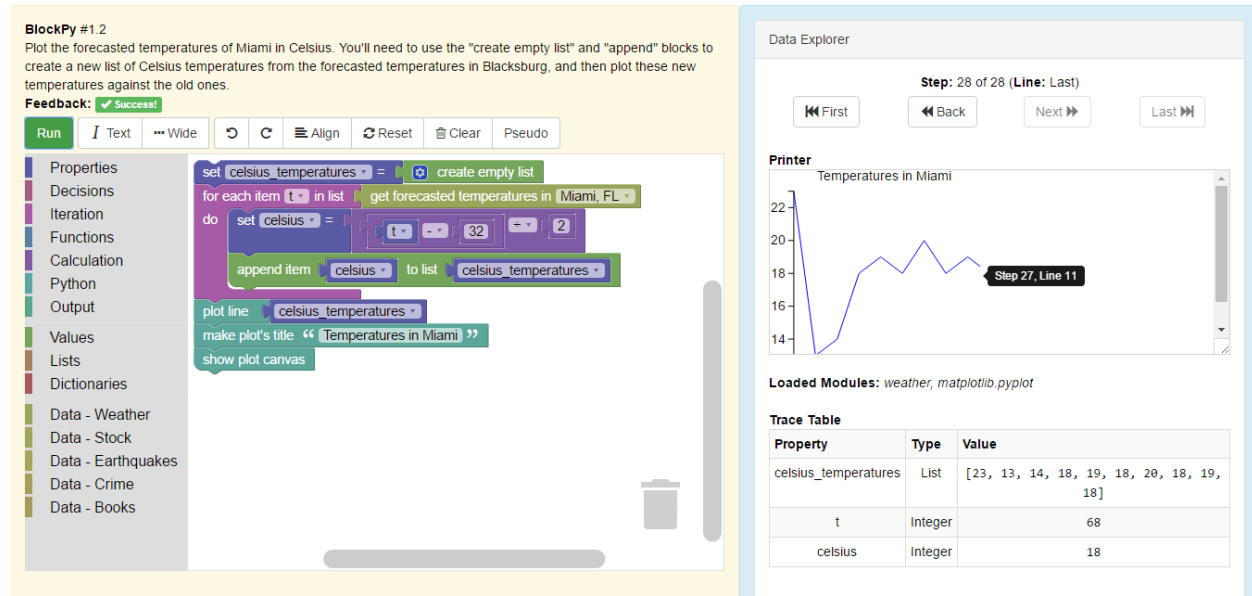


Fig. 1. A Screenshot of BlockPy in Action

the problem. BlockPy's regular error messages have been extended with additional information for beginners. 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 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 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.

**Parson's Problems** Parson's Problems are an example of how scaffolding can help a student solve a problem [34], and are available as an optional mode for a problem. When a problem is in Parsons mode, all or most of the blocks needed are provided. These blocks will be out of order, disconnected, or otherwise incomplete. The block editor will randomly shuffle the location of blocks, but it will not break their overall ordering; if statement A comes before statement B, then block A will always be above block B.

**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, simplicity, and pedagogical affordances of the data. These libraries are available through simple blocks. These



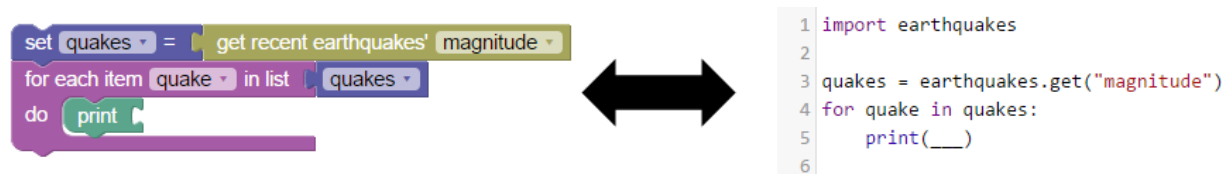


Fig. 2. Mutual Language Translation between Block and Text interfaces

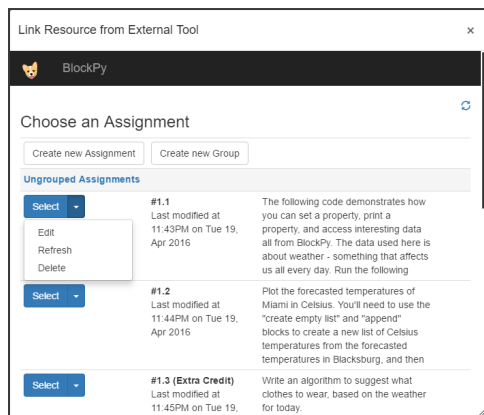


Fig. 3. Course Assignment Management

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 action.

**Plotting** Another addition to Skulpt is tools for making visualizations, a key mechanism of Data Science. 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 internal course management system, it also supports LTI (Learning Technology Interoperability). This important 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 complete problems in the environment, they are automatically graded, and the LMS is given the submission.

One service provided by LTI is that users are automatically added into the BlockPy database, authenticated 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/liti-flask-skeleton>.

**Dangling Blocks** The block editor allows users to create statements that are not fully formed or connected. At any given time, blocks may be disconnected. This is consistent with BlockPy’s design philosophy of incremental development. When a student transitions from blocks to text, unspecified expressions are converted to quadruple underscores, and unspecified statements are converted to empty “pass” statements.

**Free and Open-source** Consistent with the open-access nature of BlockPy, the entire project is free and open-source. We do not present BlockPy as a finished product, but merely another step in the development of educational tools for teaching introductory programming. We anticipate that features of BlockPy have value to other introductory environments. Making its source code freely and readily available is an important step in promoting our vision.

## 4 DESIGN ISSUES

We next present design issues that were addressed 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 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 properties to individual lines such as geometric information [26]. 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 [36] 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 this system led to BlockPy's design as a primarily text-driven environment under the hood.

## 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 more explicit 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.

## 4.3 Block Hiding

An advantage of Python as a teaching language is the terseness of its grammar. However, Python is still a language with syntax and order, and so certain commands are necessarily complex. This is not a criticism of the language, but an observation that perfectly reasonable tasks may have hidden pedagogical costs. In particular, the data science approach we take with BlockPy demands that students import modules in order to gain access to the data methods. In Python, this translates to an `import` statement and a method call on the imported module.

The goal of the Data Science context is to minimize the secondary pedagogical requirements that may distract from primary learning objectives. Therefore, in the BlockPy interface, import statements are not rendered directly in the block interface, but are present in the text view. Obvious alternatives include creating special `import` blocks that appear on demand, or a special notification within the interface that an external library is being used. Both approaches necessitate discussion of code importing.

A major disadvantage of hiding the imported module is that the module name still enters into the program's namespace. This means that students may end up choosing reasonable variable names that collide with modules that they unknowingly use. Figure 4 demonstrates code that extracts data from a module and then overwrites the module's variable with that

---

```
import stocks

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

---

Fig. 4. Degenerate Student Code

data. Although this does not have any impact on the syntactical or semantic execution of the code, the code becomes misleading. Alternative names could be used to prevent namespace collisions, such as `weather_module` or `stocks_data_source`.

## 4.4 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 from the text editor, 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 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 constructs are introduced. Although BlockPy is built on the premise that it is worth delaying the conversation about syntax, all students must eventually become 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 (partially because some students may not even understand their 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.

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

## 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.

In the Spring 2016 offering, 50 students were assigned 34 BlockPy problems over the course of five class periods (see Table 1). 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. This was their first introduction to programming in the course, coming after a 6-week introductory unit that covered conceptual knowledge of computation. 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 encouraged students to become familiar with writing code in that form. More information about the curriculum can be found in [14], and at the public site for the course ([http://think.cs.vt.edu/course\\_materials/](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 completed. 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 question about the chronological placement of BlockPy in the course, and then had

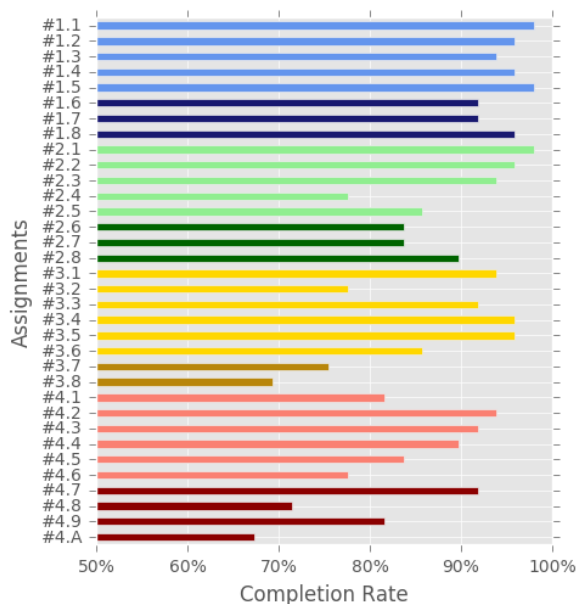


Fig. 5. Completion Rate by Problem (colors indicate distinct days, brightness indicates homework vs classwork)

four free-response questions asking students about frustrating and helpful features in both BlockPy and Spyder.

### 5.1 The State of the Curriculum

Figure 5 describes student completion rates over the assignments. Figure 6 describes the distribution of time spent on each problem. Most problems were completed by more than 75% of the students; further analysis of the log data suggests 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.

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. 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, and 7 students felt it should be earlier.

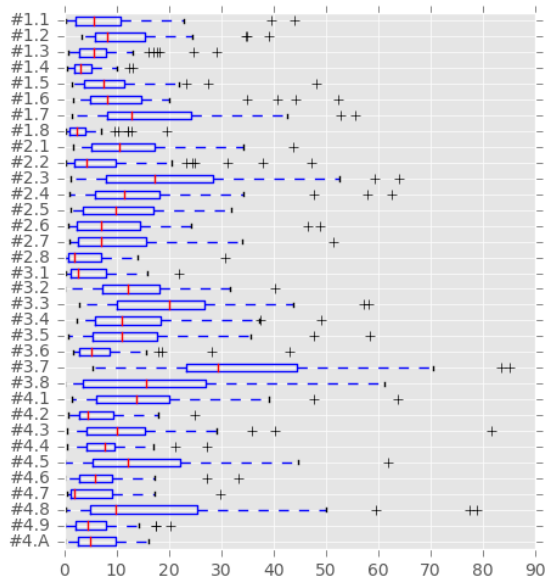


Fig. 6. Work Session Length (minutes) by Problem

## 5.2 Improved Guidance Is Needed

In the free response section, there was little agreement about what was most frustrating about Blockly, with one major exception: 34% of students agreed that Blockly's automatic guidance could be frustratingly vague. This is partially biased by the timing of the survey. Earlier problems were (somewhat) intentionally equipped with more extensive suggestions and guidance than later problems, so students were most recently working with less helpful problems. Regardless, 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 Blockly. 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 Blockly. Considering that the text mode of Blockly does have this feature, we believe that students were only considering the block interface when making this assessment. However, it is an understandable concern. Errors in Blockly refer to line numbers instead of individual blocks, a disconnect that was not considered enough by the designers.

## 5.3 Static Analysis Is Needed

Table 2 reveals a bigger concern than lack of guidance or inexact error reporting. The final submission from

**TABLE 2**  
Incidences of Semantic Errors Detected by Static Analysis

	All	Successful
Changed type of variable	959	909
Variable overwritten without read	803	755
Variable never read	265	209
Variable read without write	190	133
Iteration list used inside iteration	118	92
Iteration variable unused inside iteration	103	75
Used iteration list as iteration variable	95	86
Iteration over non-list	25	11
Used unknown function	10	9
Iteration over empty list	1	1

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, and the third column is how many incidences were found in programs marked 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.

## 5.4 The Interface has Merits

In the free response section, students were positive about Blockly's block interface. 65% indicated the block interface as particularly helpful, and 34% indicated that the dual text/block interface helped. Consistent with the criticism of the automatic guidance, only 12% found that feature particularly helpful; our interpretation from this result is that students appreciated 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 environment (e.g., students described frustrations with Python syntax, trying to interpret errors).

An interesting result from analysis of the logs is shown in Figure 7. **Edit distance** for each user modification (e.g., keystroke, block deletion/addition, etc.) was calculated and then divided by the amount of time spent in the relevant mode. From this, an estimate of how productive students were in the two different





Fig. 7. Edit Distance/Second in Text Mode vs. Block Mode

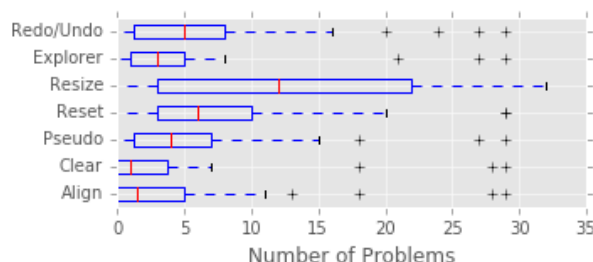


Fig. 8. Students' Use of Toolbar Features over Problems

modes can be seen. The average edit distance was more than twice as high with blocks compared to text, most likely because an action in blocks is usually equivalent to a number of keystrokes in text. Although this could easily be different for experts and should be taken with caution, it is an encouraging result for block-based programming advocates.

### 5.5 Tool Use in BlockPy

The main toolbar in BlockPy gives students access 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 educational nature of the environment, such as the code explorer and natural language code generator (named `pseudo` in this version). Figure 8 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.

### 5.6 Use of the Dual Text/Block Editor

Figure 9 shows the average length of time that students spent in the text interface compared to the block interface. Unsurprisingly, students tended to spend more of the last day in the text interface, which was encouraged by the interface starting in text mode. It is encouraging to note that students did switch between

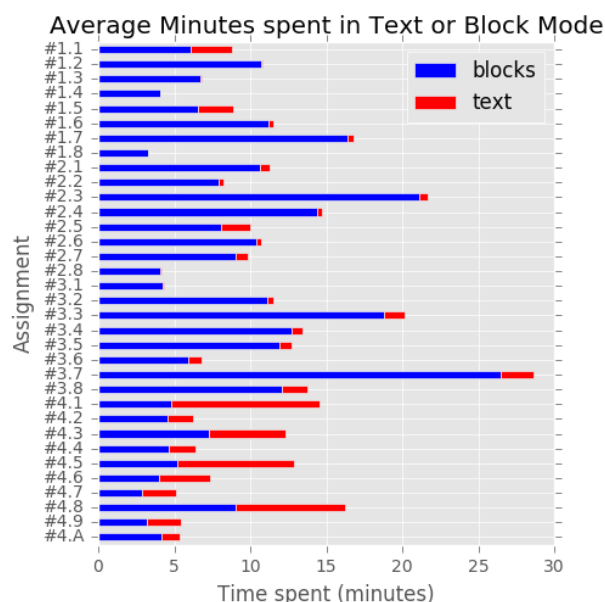


Fig. 9. Time Spent in Text vs. Block Mode

the block and text editor in several problems, if only to observe the resulting code. Deeper analysis of these results suggest that different students checked the text version of their code at different times, rather than the same students consistently doing so. Although promising, 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.

### 5.7 Misleading Blocks

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 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: 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*. Nonetheless, it does seem prudent for the guidance to correct students who might be using such incorrect code forms.

## 6 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.

### 6.1 Improved Guidance

The most obvious need for improvement in BlockPy is the automatic 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 interface to streamline types of feedback that instructors most often give. Some of these features are related to ensuring that the students' output matches expectations. 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], [38], 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, systematic 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 syntax 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 completed problems. We envision a much richer system.

First, learning resources should be made directly available as needed; for example, relevant chapters of the open-access course book or short instructional animations and videos. Second, the environment should prompt the learner to take advantage of its pedagogical tools; students may not be using features like the property explorer and natural language code explanation 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.

### 6.2 Tiered Block Interface

Transitioning students from the block interface to the text interface and eventually to a professional environment remains an unsolved problem. Although students seem to handle this fairly well, they did suggest some difficulties in the survey. We believe that establishing 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 continuous transition between blocks and text [26], and Greenfoot 3 uses a purposefully structured interface to make blocks seamlessly mimic text [27]

### 6.3 Missing Language and API Features

Although the underlying Skulpt execution environment is a full Python implementation, the entire library 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 organically.

### 6.4 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 uninteresting. Other approaches to introductory computing have their own motivational and pedagogical benefits. For instance, animation and game design have both proven to be valuable contexts, albeit with a different design philosophy. Although preliminary results gathered in our research suggest the appeal of Data Science over other contexts, our results are not conclusive, and there is not a clear disadvantage to most other contexts.

In terms of pedagogical benefits, visual programming 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 Turtle 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 assignment. 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.

## 6.5 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, students become most motivated when they are able to 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 environment with Google Sheets, so that students could access custom datasets [22]. 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. Students could be tasked with completing other phases of the Data Science process formally from within the environment.

## 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 immediate feedback. Results from an intervention with introductory students suggests ways to improve the environment. We happily make our tool freely available.

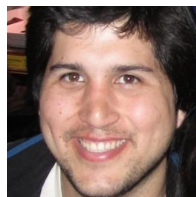
## REFERENCES

- [1] J. Wing, “Computational thinking benefits society,” *Social 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 (VL/HCC), 2015 IEEE Symposium on*. IEEE, 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*, ser. SIGCSE ’15, 2015, pp. 132–137.
- [9] 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.
- [10] B. D. Jones and G. Skaggs, *Validation of the MUSIC Model of Academic Motivation Inventory: A measure of students’ motivation in college courses*. Research presented at the International Conference on Motivation 2012, 2012.
- [11] 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.
- [12] 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*. ACM, 2006, pp. 51–58.
- [13] A. C. Bart, “Situating computational thinking with big data: Pedagogy and technology (abstract only),” in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’15, 2015, pp. 719–719.
- [14] 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, pp. 63–68.
- [15] 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*, ser. SIGCSE ’14, 2014, pp. 465–466.
- [16] 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.
- [17] P. Guo, “Python is now the most popular introductory teaching language at top us universities,” *BLOG@ CACM*, July, 2014.
- [18] R. Schutt and C. O’Neil, *Doing data science: Straight talk from the frontline*. O’Reilly Media, Inc., 2013.

- [19] 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.
- [20] 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*, ser. ICER '15, 2015, pp. 101–110.
- [21] 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.
- [22] J. D. Hellmann, "Datsnap: Enabling domain experts and introductory programmers to process big data in a block-based programming language," Master's thesis, Virginia Tech, 2015.
- [23] B. Broll, P. Völgyesi, J. Sallai, and A. Lédeczi, "Netsblox: a visual language and web-based environment for teaching distributed programming," 2016.
- [24] 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. IEEE, 2015, pp. 71–75.
- [25] Y. Ohshima, J. Mnig, and J. Maloney, "A module system for a general-purpose blocks language," in *Blocks and Beyond Workshop (Blocks and Beyond)*, 2015 IEEE, Oct 2015, pp. 39–44.
- [26] 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.
- [27] 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.
- [28] S. H. Edwards, D. S. Tilden, and A. Allevato, "Pythy: Improving the introductory python programming experience," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14, 2014, pp. 641–646.
- [29] 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.
- [30] 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*, ser. SIGCSE '13, 2013, pp. 579–584. [Online]. Available: <http://doi.acm.org/10.1145/2445196.2445368>
- [31] S. Graham, "Skulpt," 2010.
- [32] N. Fraser *et al.*, "Blockly: A visual programming editor," URL: <https://code.google.com/p/blockly/>, 2013.
- [33] M. Haverbeke, "Codemirror," 2011.
- [34] D. Parsons and P. Haden, "Parson's programming puzzles: a fun and effective learning tool for first programming courses," in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 2006, pp. 157–163.
- [35] M. Ebert, "Skulpt matplotlib," [https://github.com/waywaaard/skulpt\\_matplotlib](https://github.com/waywaaard/skulpt_matplotlib), 2014.
- [36] 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*, ser. SIGCSE '15, 2015, pp. 185–190.
- [37] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 15–26.
- [38] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: a self-improving python programming tutor," *International Journal of Artificial Intelligence in Education*, pp. 1–28, 2015.



**Austin Cory Bart** is a PhD candidate at Virginia Tech studying Computer Science and 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** is a PhD candidate at Virginia Tech studying Computer Science. He has an undergraduate degree from Escuela Superior Politécnica del Litoral (ESPOL), and a MSc from Universitat Politècnica de Catalunya. His research interests are in Human-Computer Interaction, and Computer 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 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.