Implementing an Open-access, Data Science Programming Environment for Learners

Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A. Shaffer, Dennis Kafura
Computer Science
Virginia Tech
Blacksburg, Virginia, USA
acbart@vt.edu, jtibau@vt.edu, tilevich@vt.edu, shaffer@vt.edu, kafura@vt.edu

***Abstract*—A key retention issue when educating computing novices is ensuring that the frustrations of mastering programming fundamentals do not demotivate students. Non-CS majors often struggle to find relevance in traditional computing curricula that tend to either emphasize abstract concepts, focus on non-practical entertainment (e.g., game/animation design), or rely on decontextualized settings. To assist with these issues, this paper introduces BlockPy (http://www.blockpy.com), a web-based, open-access Python programming environment that supports introductory programmers in a Data Science context. It promotes long-term transfer by scaffolding an introduction to textual programming through a block-based programming view, ideal for beginners. BlockPy is designed for informal learners and formal classes, and provides guiding feedback within interactive programming problems. The results from a pilot of the initial deployment of BlockPy indicates that the environment addresses many problems faced by novice learners.**

***Keywords*-Computer science education; Computer aided instruction; Data analysis; Web services;**

As computing becomes pervasive in our society across fields, working professionals increasingly need some expertise in computing alongside their core domain knowledge. General education computing curricula at the university level (e.g., "Computational Thinking" courses) are scaling, Massively Open Online Courses are flourishing, and a large class of learners are pursuing non-formal learning experiences on their own. Both these traditional and non-traditional learners often have little experience with computing, low self-efficacy, and are uncertain how computing benefits their long-term career goals. Not only do they need special scaffolding unique to their ability and motivational level, but they also need fewer barriers in accessing these materials. Our new tool to better serve this population is BlockPy: an open-access, web-based Python environment for data science that supports learners with guided instruction and an accessible interface (http://www.blockpy.com).

Why Data Science? Modern approaches to contextualizing introductory courses have focused on making the experience "fun" and "interesting", with an emphasis on game design and media computation. However, student motivation is a complex construct dependent on more than just situational interest; holistic models of motivation suggest that students also need to feel that the material is useful to learn, and that long-term career goals are satisfied [1]. Contexts like Media Computation are not always perceived as authentically useful for non-majors, based on a study by Guzdial et al [2].

We suggest that Data Science is a motivating context that can appeal in a different way to students, thanks to the wide-spread need for data processing in other majors. In a previous work, we have reported on the affordances and impacts of Data Science as a learning context [3]. Often, students study computing to learn how to manage the dizzying quantities of data being stored and analyzed in a discipline or for a specific self-derived project. By grounding the content in this context, students can be more easily convinced of the relevance of computing and understand how the materials fit together more clearly. By aligning the context with students' long-term needs, students also learn skills more relevant to those needs. Finally, data science as a context naturally lends itself to teaching topics related to structured data, iteration, and other core material, making it a pedagogically valuable context to the CS instructor.

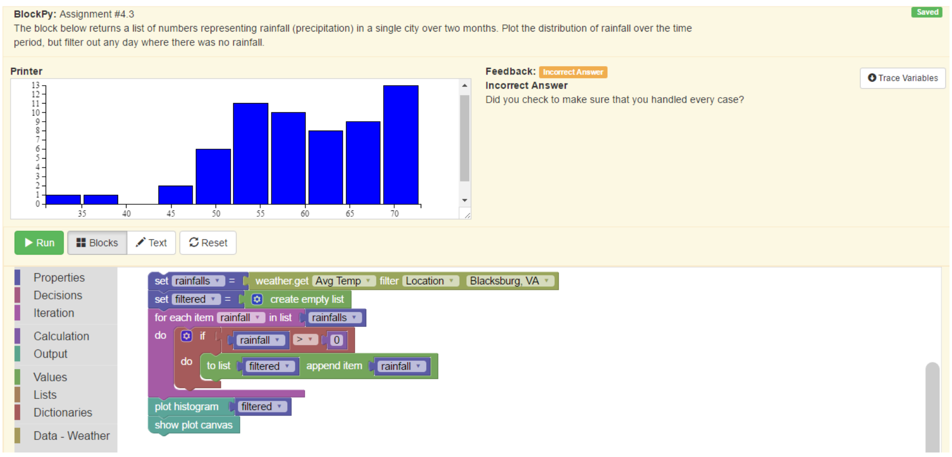
Why Python? Python has become one of the most popular introductory programming languages [4], thanks to its simple syntax combined with impressive power. It includes strong support for data science thanks to popular libraries like MatPlotLib. Python requires little code to accomplish interesting things, so novices are not bogged down with complex syntactical details. Its wide-spread use in both introductory classes and industry further motivated our choice.

Why Blocks? Any kind of programming is a challenge to beginners, due to the nature of coding as the "most powerful, but least usable human-computer interface ever invented" [5]. Block-based languages (such as Scratch and AppInventor) have been shown to mitigate the start-up time for students to begin programming and accomplish tasks [6], [7]. By providing structure and an immediate view into the entire user interface of a language, blocks greatly benefit introductory learners.

Why Another Python Web Environment? There are several environments available today that let students and instructors write Python in the browser, including CodeSkulptor [8], Pythy [9], and the Online Python Tutor [10]. BlockPy stands on the shoulders of giants, integrating features inspired by these environments and introducing novel ones. But none of these existing Python environments transitions students into textual programming languages.

BlockPy was designed to provide dual support for both block-based and text-based code authoring. At any time, the student can switch freely between a block-based view of their code and a traditional text-based view. This powerful feature is inspired by Pencil Code, which uses its own Logo language [11], and similar implementations have been successful as a fading scaffold for students [12].

BlockPy extends Pythy's [9] support for "assignments", problems that integrate presentation with assessment. However,



BlockPy: Assignment #4.3
The block below returns a list of numbers representing rainfall (precipitation) in a single city over two months. Plot the distribution of rainfall over the time period, but filter out any day where there was no rainfall.

Feedback: Incorrect Answer
Incorrect Answer
Did you check to make sure that you handled every case?

```
set rainfalls = weather.get Avg Temp filter Location Blacksburg, VA
set filtered = create empty list
for each item rainfall in list rainfalls
do  if rainfall > 0
    do  to list filtered append item rainfall
plot histogram filtered
show plot canvas
```

Figure 1. BlockPy in action

Pythy only supports traditional unit testing to provide students with feedback, while BlockPy provides an API for code analysis and free-form text guidance that instructors can configure to give helpful suggestions to their students. Furthermore, Pythy has limited support for data science, whereas BlockPy has a rich library of data sources and a MatPlotLib-based plotting API.

CodeSkulptor, Pythy, and BlockPy all use the same internal engine for running Python code ("Skulpt"). Although CodeSkulptor has an extensive API for creating user interfaces and games, it provides a non-standard library. Although suitable for beginners, this library does not aid the transition to serious programming environments. In BlockPy, the philosophy is to maintain approximate compatibility with real systems. Instead of a custom plotting API, for instance, we mimic the MatPlotLib interface.

OnlinePythonTutor has proven to be a useful tool for visualizing program state. However, OPT gives a depth of detail that can overwhelm introductory students (e.g., terminology such as Frames and Objects, which might be foreign to students). BlockPy's state explorer does not attempt to match OPT's thoroughness, but instead provides a helpful yet simple picture of program state. Additionally, we avoid OPT's server dependency by relying on Skulpt, which runs in the browser.

I. BlockPy

The primary design goals of BlockPy are as follows.

1) Reduce barriers to learning programming.
2) Promote authenticity by empowering students to complete real-world problems.
3) Promote maturity by faded scaffolds (e.g., transitioning from blocks to text).

4) Minimize the need for help from human instructors.

A. *Open-source, Open-access*

The fundamental vision of BlockPy is a highly accessible, web-based platform for anyone to learn how to program. All code is open-source, and leverages a number of open-source libraries. There is no registration needed to use the software, although there are features that benefit from free registration, such as managing classrooms. We provide guided learning materials, to be shared by educators.

The BlockPy editor continuously stores user code as it is entered. Logs are stored at the keystroke level for future program analysis (described in Section I-E). The latest version of the user's code is therefore available between sessions. When operating in offline mode, the code is stored in the `LocalStorage` browser object; when the connection is reestablished, synchronization is performed.

B. *Python Execution*

The BlockPy system is built to work offline, ideal for places where internet connectivity is unreliable. Python Code Execution is achieved through a modified instance of the Skulpt JavaScript library. Skulpt is a full Python parser and compiler, supporting almost all Python language features by generating JavaScript code. This includes partial support for the rich Python standard library. The Skulpt execution environment resides entirely within the users' browser, so there is no reliance on an external server beyond the initial page load.

C. *Block-based Python*

To support introductory learners as they grapple with Python syntax, the initial interface in BlockPy is block-based, using the popular Blockly JavaScript library. Language features (iteration, decision, variable assignment and access, etc.) are contained in a toolbox on the left side, from which users drag-and-drop blocks onto a canvas. BlockPy's block interface only generates syntactically valid Python code, enforced by the "snapping" connectors of the blocks (although it is possible to generate semantically incorrect code – discussed later). This block interface is synchronized with a text interface; section I-H describes these two interfaces.

An important question is how many language details should be exposed, and at what rate. A rarely used feature of `for` loops in Python is to contain an `else` clause that is executed upon successful completion of the loop (that is, when it is not prematurely escaped using a `break` statement). This advanced language feature is similar to a `finally` statement with exceptions. However, if an `else` clause were made available to beginners first trying to grapple with iteration, it is likely they would confuse the concept with the conditional `else` clause used in `if` statements. Cognitive load is harsh to beginners, and the user interface needs to avoid exposing unnecessary details where possible. While hiding `else` bodies in a `for` loop is a clear case, there are more subtle examples. It can be difficult to recognize when the learner is ready to use parallel assignment, and therefore should be able to specify multiple variables on the left side of an assignment block. A block-based language forces a teacher to make important decisions about how to expose language features. As part of the future work of BlockPy, we are experimenting with exposing language features at different rates, adjustable by the instructor, so the system can expose a progressively more accurate language model.

D. Adaptive Guided Practice

One of the most powerful features of BlockPy is the interactive, guided feedback feature. A limitation of programming environments like Snap! is that they are not pedagogically interactive – students completing an assignment in the system are not guided to success. The learner must decide when they have completed their program, and whether it meets the specification. For independent learners outside of a formal learning experience, this requires high levels of self-regulation and meta-cognition. BlockPy's adaptive elements follow an Expert model; when students run their code, it is checked against instructor-provided logic. If the student code fails for some reason, they are offered a suggestion. Correct code gives a green "Complete" mark – we have found that this positive marker has motivating power.

In the Instructor Mode, teachers provide a problem instance. First, a WYSIWYG rich-text editor edits the problem description, supporting any valid HTML content (e.g., images, links). Second, the instructor provides code in special canvases that affect the students' experience. Instructors write this code using the same text/block interface that students use. First is the "starting code", shown to the student when they begin the problem, avoiding a blank canvas. The second instructor code defines interactive feedback, which can access the students' code, their final output, and a complete trace of their program's state. The checking system can declare the code to be correct, or display an HTML string that is rendered as user feedback. The instructor is free to write whatever logic they want, such as searching for a specific Abstract Syntax Tree (AST) element, testing the outputs on the console, or walking through the programs state to satisfy invariants. An API for common checks is evolving based on common use cases, such as parsing the program's AST to ensure that they are not calling forbidden built-in Python functions. We are also exploring interventions an instructor can make beyond rendering textual feedback; perhaps displaying a pop-up dialog with an embedded instructional video, or alerting an instructor to provide Just-In-Time instruction for a particular struggling student. Finally, we are experimenting with a new system for modeling students' misconceptions related to programming, in order to provide better guidance tuned to students' specific mistakes.

E. Program Analysis for Deeper Learning

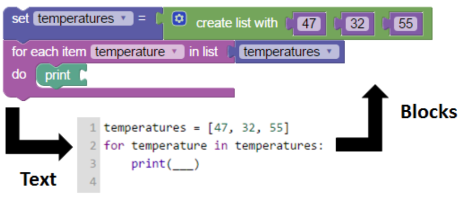
BlockPy uses simple program analysis techniques to find both general mistakes that novices make, and problem-specific errors. For example, beginners often fail to understand the true purpose of certain variables, and incorrectly include them due to mimicking an example. By performing simple variable liveness analyses, we can identify these variables and raise an error. Most modern editors feature this kind of analysis, but usually trust that the user has a reason and responds passively; the nature of our learners requires stronger responses.

Since we securely record the history of users' programs and log interface interactions, we can mine this repository of code to infer common patterns that suggest undesirable learner behavior. For example, users that frequently move the same blocks without progressing in the problem objectives might be indicative of taking longer on the problem than other users. Alternatively, students who pick a decision block to complete a problem about iteration might need extra attention. By proving or disproving such hypotheses, we can improve the automatic feedback of the system and provide more individualized support.

F. LTI Support

BlockPy supports the LTI (Learning Technology Interoperability) protocol. This is a mechanism by which instructors can embed questions in their existing course management software (e.g., Canvas, OpenEdX) and receive assignment outcomes (e.g., a grade).

A typical learner uses BlockPy without ever being aware of LTI. An instructor using the system obtains a secret key and configuration URL that is used within their Learning Management System. Students on a course website may use BlockPy without registering for a BlockPy account – the first time they log in through their LMS' provided link, they are invisibly registered in the system


set temperatures =
create list with 47 32 55
for each item temperature in list temperatures
do print
Blocks
Text

```
temperatures = [47, 32, 55]
for temperature in temperatures:
    print(___)
```

Figure 2. User perspective of block/text transition

with a regular account through additional information from the LMS. As students complete work, assignment progress is reported back to the LMS. Instructors use a special interactive menu for managing exercises associated with a course.

G. Data Science Blocks

BlockPy focuses on Data Science as its primary context, and so we have blocks for working with data. We support a subset of the popular MatPlotlib library, and extend it with connections to the CORGIS project [13]. The MatPlotLib API, for example, provides a "plot(list)" function to create simple line plots. By mimicking MatPlotLib, students can seamlessly shift to a serious Python programming environment without loss of code.

The CORGIS (Collection of Real-time, Giant, Interesting, Situated datasets) Project makes motivating datasets available to introductory students through simple programming libraries [13]. These datasets are drawn from many disciplines, resulting in material meant to be universally interesting and relevant. Currently, BlockPy supports a number of different CORGIS libraries including weather data, earthquake data, United States crime statistics, and a classic book dataset.

H. Mutual Language Translation

A technical contributions of this project is the mutual language translation between Blockly and Python. Blockly outputs valid python source code, which can be passed into Skulpt to extract a JSON representation of the Abstract Syntax Tree. This AST is parsed using our own Py2Block library to generate an XML representation that Blockly can render in the Block View. Figure 2 demonstrates the users' experience. When a student tries to convert code with disconnected blocks, the generated Python code will be filled in with triple underscores. These underscores (usually a valid variable name in Python) will trigger a run-time error.

Blockly already supports compilation of its blocks to Python, JavaScript, PHP, and Dart. However, this multiple language support causes reduced isomorphism—each language has different syntax for their common operations, and it is impossible to create a fully-featured block language with a one-to-one mapping to them all. For example, JavaScript has no support for parallel assignment, a commonly-used feature in Python, while Python does not have a unary increment operator. Blockly itself has syntax and vocabulary descended from Logo.

Instead of trying to satisfy multiple languages, we have dropped support for other languages for a more fully-featured mapping to Python. This requires minor changes that introduce Python-centric syntax details: function blocks are labeled "define", assignment blocks have an "=" symbol, the "add item to list" block is renamed to "append". Blockly has also been extended with new language features, including dictionary access and creation.

Eventually, the interface should offer a complete isomorphic mapping to Python. However, there are a number of complications to resolve first. For instance, Python uses square brackets for both list indexing and dictionary access. There is a strong desire to differentiate between these types of access, visible in the block view as two distinct kinds of blocks ("get ith element of list" vs. "get key from dict" blocks). However, it is computationally difficult to statically identify the usage of a given pair of brackets– sophisticated program analysis techniques are needed.

I. Parson's Problems

Parsons' Problems are a special type of coding exercise where all of the necessary code blocks are present, but disconnected and shuffled. These kinds of problems scaffold beginners by providing everything they need to complete the problem, reducing many of the barriers to getting started. BlockPy supports these types of problems with a special "Parsons Mode" where top-level blocks are shuffled in the block mode.

J. State Explorer

BlockPy provides a State Explorer, used to trace programs' execution over time. The State Explorer displays more than just information about variables: Users can step through the code's execution, affecting what is currently printed/-plotted, imported modules, and the values and types of variables.

II. Model Use Cases

In this section, we consider some example scenarios that describe our vision of typical BlockPy use cases. Our intent is for BlockPy to be useful in both formal and informal situations.

A. Independent Learner

A learner independently logs into the BlockPy system and selects an introductory problem on calculating averages using iteration: *"Is the weather in Seattle above 60 degrees Fahrenheit? Print Yes or No."* As a complete novice, they are unsure what to do after reading the problem description. If they decide to cheat by checking the current weather in Seattle and printing the literal value, the system intelligently notices that they are missing a relevant weather block, and explains that they need to combine programmatic decision logic with the appropriate data source. They think to access the "Weather" block category, and grab the `weather get` block, but are unsure what to do next. When they run their program, the system notices that they have not used any `IF` statements, and suggests reading a linked chapter in an online textbook. If they continue to struggle with integrating pieces, the system can provide increasingly detailed hints until they succeed.

B. Classroom Lesson

Another common use case for the system would be an instructor with a large classroom of students. The instructor is using Canvas, an LTI-capable LMS. They create a series of assignments for the day's classwork. Students log into Canvas and begin working on the assignments. As they complete the assignments, their grade is reported to Canvas. The instructor can monitor progress for the class and check which students are struggling to complete assignments. This information can allow them to target under-performers with earlier interventions. The more automatic feedback that instructors make available, the less they need to focus on simple problems ("You were checking the temperature for the wrong city.") and the more they can focus on students that are truly struggling ("What is iteration?").

C. 1-1 Tutoring

On several occasions, we have found BlockPy to be a useful tool for correcting individual students' misconceptions. In particular, the block representation of programs can help beginners grasp that code is not a series of symbols but a structured representation of an algorithm. Consider a student struggling to write the necessary syntax for indexing a nested dictionary (e.g., a crime report broken into multiple levels, with the burglary rate for a city nested under a violent crime categorization). The student may not have a clear image of how the layered structure of data can translate into a chain of dictionary accesses. Sitting with the student, the instructor could build up an expression accessing the data by connecting together dictionary access blocks to the data block, showing the generated Python code at each step. The learner can visually see how chunks of the code correlate to blocks.

III. Pilot Study

BlockPy was piloted in an introductory Computational Thinking course with

35 students in Spring 2015. These students come from a diverse range of majors, including liberal arts (57%), architecture (17%), and sciences (15%). There were 20 female students (57%) and 15 male. The vast majority of students reported no prior experience in programming, less than 17% having taken the high school AP course. Students were evenly distributed across years, with slightly more seniors (29%), equal percentages of sophomores and juniors (26%), and fewer freshmen (14%).

The course content focused on teaching Abstraction and Algorithms. While programming was not a primary learning objective, was is an important topic in the course for concretely talking about higher level objectives. The first third of the course, students worked with NetLogo (although they do not program in it, they do read code) and participated in explanatory kinesthetic activities. Then students were introduced to Python using BlockPy, where they spent roughly six classes on completing guided practice problems. The next two classes were devoted to using a regular Python environment (Spyder) to complete small programming assignments (similar to the ones done with BlockPy). Finally, students were given eight class periods to work on an individual final project in Spyder.

A. Methodology

Student responses to BlockPy were collected through two surveys, one given after the BlockPy section and the other given at the end of the course. The survey was composed of 4-point Likert questions and open-ended qualitative questions. A selection of particularly interesting results are shown in Figure 3. All conclusions from this study should be considered preliminary, since it was with the first version of the BlockPy environment.

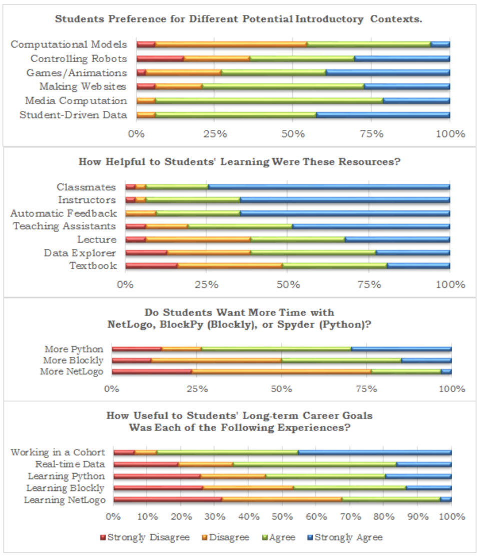


Figure 3. Student Responses from Survey on BlockPy

B. Perceptions of BlockPy

The first survey question asked whether students wanted more time with each of the programming environments they used in the course: NetLogo, BlockPy, or Spyder. Note that BlockPy was referred to as “Blockly”, and the Spyder environment was referred to as “Python”. These results suggest that students valued their experiences with BlockPy more than NetLogo, but mostly felt that they were not getting enough Python experience. This is backed up by the qualitative data, where some students say “More Blockly, Less

Python", but others ask for "More Blockly and More Python".

C. Usage of BlockPy

Over the six days spent using BlockPy, students were tasked with 40 classwork questions and 19 homework questions. Students ran their code an average of 4 times per problem (standard deviation 1.8).

Students were asked if they felt successful in the transition from BlockPy to Spyder. Only 65% of the class agreed or strongly agreed, suggesting that there was a sizeable population that felt uncomfortable during that transition. The original design of the mutual language translation featured the block and text view simultaneously, side-by-side. However, analysis of the logs reveals that most students did not take advantage of the feature. Only 5 students (roughly 15%) had used the conversion functionality at all, and fewer used it consistently. It is possible that students were observing the code as it changed, but they were not writing textual code. It is difficult to say why exactly students did not take advantage of it. Our current hypothesis is that students were confused by the interface, which required manual conversion to go from text to blocks. In our new version, the conversion happens automatically, simply by switching tabs, and we provide intentional opportunities for the students to switch. Preliminary data suggests this new interface greatly improves students' transition.

Students were surveyed about what helped their learning the most. Peer learning and instructors were about on par with the automatic feedback given in BlockPy, suggesting the strong value of the system. Despite the popular response to the State Explorer, relatively few students took advantage of it (11 students, roughly 31%). Since more than 50% of the class reported finding value in the data explorer, it is possible that the students benefited from instructor presentations of the tool, even if they didn't take advantage of it themselves.

D. Data Science Context

Students were surveyed about their perceptions of the value of different course experiences with regards to their long-term career goals and their interest in potential contexts for introductory computing courses. Each of these contexts were briefly described – for example, the Media Comp context was listed as "working with pictures, sounds and movies." Both sets of results suggest that students find data science to be compelling, but this should be taken with a grain of salt, since students have negligible experience with alternative contexts. However, our preliminary results suggest that this is an approach worth exploring further.

IV. Future Work

BlockPy is an evolving project. We have a number of features planned to expand Python support. We are also planning on expanding support for the guided feedback API for instructors, such as leveraging more static/dynamic type inference techniques to improve block rendering and error reporting.

We also have research questions posed by the block-based nature of the interface.

One of the biggest values of a block-based environment is that it can immediately expose the breadth of a rich API. This greatly reduces students' dependency on documentation. Of course, exposing this breadth can also be a downside, as students might be overwhelmed by the features in the interface. It is an open research question to decide what rate to expose language features.

One of the major advantages of game and animation design as an introductory context is that they make abstract concepts concrete. Further analysis is needed to determine the trade-offs of using different contexts. BlockPy can support this by supporting these alternative contexts, such as turtle graphics and media computation libraries.

It is difficult to derive conclusive results from our pilot due to the small population size and the evolving nature of BlockPy. Preliminary results from more in-progress studies suggest that recent improvements have overcome a number of limitations to the environment and user feedback has dramatically improved. We are conducting follow-up studies on the logged students' code, even as we collect more data on the newest iteration. We are hopeful that BlockPy will increase its user base, providing a larger sample of learners to conduct research on, and provide more meaningful data.

A. *Missing Language features*

BlockPy is being developed in an on-demand fashion, driven by immediate course needs, but is still limited. For example, the block interface does not support a number of advanced Python features, such as an interface for writing Object-oriented classes. This does not mean that students cannot write programs featuring classes or other advanced features. Python code using these features will render in BlockPy as embedded text blocks and will execute through Skulpt normally. There is no technical impediment to supporting these features, the process is limited only by time and community interest.

V. Conclusion

In this paper, we have introduced BlockPy, our block-based environment for Python. It is open-source and available for use for free at http://www.blockpy.com/. We believe that BlockPy represents a new paradigm for introductory learners, blending interactive support with a strong path to programming maturity. By teaching in the context of data science, we provide authenticity even as we move students out of the system towards a more serious environment. Research with BlockPy will help answer crucial questions about the value of data science and blocks. Our hope is that BlockPy's open nature can encourage learners from diverse fields to engage with computing in a way that will lead to a computing-rich future for a larger population.

Acknowledgment

We gratefully acknowledge the support of the National Science Foundation under Grants DGE-0822220, DUE-1444094, and DUE-1624320.

References

[1] 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.

[2] 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.

[3] A. C. Bart, R. Whitcomb, E. Tilevich, C. A. Shaffer, and D. Kafura, "Computing with corgis: Diverse, real-world datasets for introductory computing," in *Proceedings of the 48th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '17, 2017.

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

[5] A. Ko, "Programming languages are the least usable, but most powerful human-computer interfaces ever invented," http://blogs.uw.edu/ajko/2014/03/25/programming-languages", 2014.

[6] 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.

[7] 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.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[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.