JupyterLab Extensions for Blocks Programming, Self-Explanations, and HTML Injection

Andrew M. Olney
University of Memphis
365 Innovation Drive, Suite 303
Memphis, Tennessee 38152
aolney@memphis.edu

Scott D. Fleming
University of Memphis
Dunn Hall 375
Memphis, Tennessee 38152
Scott.Fleming@memphis.edu

ABSTRACT

JupyterLab is a widely used platform for programming and data science using computational notebooks, but it has not been widely used in the educational data mining community as a source of student data. We have developed three JupyterLab extensions to enable educational data mining research in CSEd and data science. Our Blockly extension supports blocks-based programming in JupyterLab and logs both event-level blocks actions as well as kernel actions and errors. Our self-explanation extension appends self-explanation prompts to codes cells and logs the input text for further analysis. Finally, our HTML injection extension allows injection of arbitrary HTML and Javascript into JupyterLab notebooks to enable pedagogies and data collection currently unsupported by JupyterLab. All extensions are open-source and distributed through NPM.

Keywords

JupyterLab, blocks programming, self-explanations, process data, HTML injection

1. INTRODUCTION

Computational notebooks have been adopted by professional data scientists [8], scientists generally [18], and are becoming increasingly popular in computer science education [7, 12, 20]. The popularity of computational notebooks stems from their ability to combine text, mathematical equations, code, and graphs. By combining these elements, computational notebooks allow data scientists to create shareable, *reproducible* reports: anyone receiving a computational notebook can recreate the original analysis or modify it to ask new questions. Like any report, a computational notebook contains text explaining each step and describing results. O'Hara et al. summarize it well: "A computational notebook is a document that can be read like a journal paper and run like a computer program."[12, p. 263]

`JupyterLab` is the current iteration of the open-source Project Jupyter and is widely used, with over a million computational notebooks on GitHub [17]. The name Jupyter is a portmanteau of the programming languages Julia, Python, and R, which were the original target languages of Jupyter, but now dozens of languages are supported and can operate simultaneously within a single notebook [14]. The success of Project Jupyter has seen it integrated in a variety of platforms, including Google's Colaboratory, Kaggle's Kernels, and Microsoft Azure Notebooks, and as a result, Project Jupyter is likely the most widely used computational notebook today.

`JupyterLab`'s notebook frontend is web-based and presents the user with a top-level menu followed by an expanding list of cells. Each cell can be text, code, or multimedia output. For example, if the last line of a code cell produced a matrix, then the following output cell would be formatted as a table, and if the last line generated a graph, then the output cell would be the graph rendered as an image. Each cell is runnable and re-runnable, and implicitly references the context of previously executed cells. This ability to chunk pieces of code is an advance over traditional interactive programming where statements are entered line by line at the command prompt, because it allows larger chunks to be created and run at once.

`JupyterLab` has what is commonly described as a plugin architecture, which makes it possible to modify the behavior of `JupyterLab` without changing its source code. In `JupyterLab` terminology, these plugins are called `JupyterLab` extensions. An extension is a software library, written in JavaScript, that extends the functionality of `JupyterLab`.

Despite its wide professional use and use in classrooms, JupyterLab hasn't been used as a source of student educational data in research published in educational data mining (EDM) conferences and journals. In this paper we present three JupyterLab extensions to advance EDM research. Our Blockly extension supports blocks-based programming in JupyterLab and logs both event-level blocks actions as well as kernel actions and errors. Our self-explanation extension appends self-explanation prompts to codes cells and logs the input text for further analysis. Finally, our HTML injection extension allows injection of arbitrary HTML and Javascript into JupyterLab notebooks to enable pedagogies and data collection currently unsupported by JupyterLab. All extensions are open-source and can be installed at the command line with the standard `jupyter labextension install` command. They may be used independently, simultaneously, or merely as models for future extensions supporting research in EDM.

Copyright ©2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR-WS.org/Vol-3051/CSEDM_8.pdf

2. BLOCKLY EXTENSION

In the last decade, blocks languages have seen wide adoption for teaching introductory programming [2, 16] as they have shown multiple positive effects on learning, including both cognitive and motivational effects, in introductory undergraduate courses [1, 5, 9, 10, 15]. Blocks languages compose code elements via irregularly shaped graphical widgets, similar to puzzle pieces or LEGO®. Their design typically makes syntactic mistakes difficult or impossible because the widgets cannot fit together in nonsyntactic ways. Furthermore, since blocks are visually browsable on an interface palette, students need only recognize them rather than the more difficult task of recalling code, cf. [19].

`Blockly` is an open-source JavaScript library for creating blocks-based editors for programming languages within a web browser [6]. `Blockly` supports five languages out of the box, including JavaScript, Python, PHP, Lua, and Dart, and compiles a given assemblage of blocks into any one of these languages through code generators. A variety of other blocks-based projects use `Blockly`, including `AppInventor`, Microsoft's `MakeCode`, and `Code.org`.

`Blockly`'s user interface minimally consists of a workspace for arranging blocks and a toolbox, or palette, for introducing blocks to the workspace. Within the blocks workspace, blocks can be dragged, copied, pasted, deleted, or snapped together. The blocks themselves contain elements like free-text entry fields and dropdowns, and dropdowns can be set to dynamically populate, e.g. with a list of current variables, rather than solely being static. Variable and function categories of the toolbox are also dynamic, such that as a variable is created, blocks for getting, setting, and similar operations are dynamically created. Likewise the function category of the toolbox yields blocks for functions which, once defined, are dynamically added to the toolbox so they can be called. To make the creation of new blocks and blocks languages easier, `Blockly` also provides a web-based graphical authoring tool for blocks that allows authors to create, modify, and save blocks configurations, including code generation.

We have integrated `Blockly` with `JupyterLab` by building a `JupyterLab` extension. When the user selects the extension, it by default opens side-by-side with the active notebook as shown in Figure 1. When a user arranges blocks in the `Blockly` workspace and then presses the `Blocks to Code` button, the corresponding Python code is generated in the active cell in the notebook, along with a serialized XML string in a code comment. The XML string allows the user to reconstruct the blocks workspace used to generate the code by clicking the `Code to Blocks` button.

Because the workspace can rapidly fill up with blocks, we have introduced a feature called notebook sync. When notebook sync is activated, clicking on a cell with an XML comment clears the current workspace and replaces it with the workspace that generated the code in that cell. This sync action is equivalent to the user manually deleting all the blocks in the workspace, selecting the target cell, and pressing the `Code to Blocks` button. This feature allows users to focus on the blocks being used in their current workflow without having to be distracted by blocks they have already used. Notebook sync brings the experience of working in `Blockly` closer to the experience of working in `JupyterLab`, where each cell can be manipulated independently of other cells.

Table 1: Blockly Extension Logged Data

Source	Name	Payload
JupyterLab	execute-code	code executed
JupyterLab	execute-code-error	message / stack trace
JupyterLab	active-cell-change	contents of active cell
JupyterLab	xml-to-blocks	xml string
JupyterLab	block-to-code	code / xml string
JupyterLab	notebook-changed	new notebook name
Blockly	block-create	event object
Blockly	block-delete	event object
Blockly	block-change	event object
Blockly	block-move	event object
Blockly	var-create	event object
Blockly	var-delete	event object
Blockly	var-rename	event object
Blockly	ui-selected	event object
Blockly	ui-category	event object
Blockly	ui-click	event object
Blockly	ui-commentOpen	event object
Blockly	ui-mutatorOpen	event object
Blockly	ui-warningOpen	event object
Blockly	ui-theme	event object

We have also introduced a feature called *intelliblocks* [13]. Intelliblocks are blocks that are dynamically configured by querying the kernel for string completions and variable information (i.e. intellisense queries). Intelliblocks first query the variable named by the block for type information, e.g. `pd` in Figure 1, and then query all the children of that variable for both completions, e.g. `pd.`, and type information. Intelliblocks appear to solve the block authoring problem of `Blockly` and allow it to be scaled up to arbitrary libraries. Without a solution like intelliblocks, a human author would be required to make thousands of blocks for a library of sufficient size, and a the user would then need to navigate through all these blocks to find the ones needed. Through our extension, intelliblocks allow research on blocks-based programming without any additional authoring effort.

The extension logs both Jupyter and Blockly process data. The logging is controlled by three query string parameters that may be appended to JupyterHub links distributed to participants: `log=xxx`, which enables logging to the specified url endpoint via POST requests; `id=xxx`, which logs with the specified participant identifier; `bl=1`, which sets the Blockly extension to auto-open in split-pane view. Each datum is logged in JSON format with a payload that is either a string or a JSON object. In the case of JupyterLab data, the format is fairly simple, as shown in Table 1. However, in the case of Blockly, the data is rather dense and complex. For example, a move event includes the block id, the previous x/y position, and the current x/y position, in addition to ids for the larger group of attached blocks and current blocks workspace. Because of this complexity, the Blockly events are logged exactly as they appear in execution, again in JSON format.

3. SELF-EXPLANATION EXTENSION

The self-explanation effect is a well-known and studied effect where asking a student to produce self-explanations enhances learning [3, 4, 11]. Self-explanation prompts elicit unstructured input from students about their thinking, and so represent a rich source of data for understanding their thought processes. Unlike the Blockly extension described in Section 2, there is no need for a side pane with self-explanations; rather it is more parsimonious to prompt for self-explanations within the notebook itself.

We have create a self-explanation extension that automatically adds self-explanation prompts to each code cell as shown in Figure 2. Below each prompt is a text entry box for the student's explanation and a button to save their explanation. While the student types, the font of the text is red, until they press the save button, at which point it is logged and the text changes to black. Similar to the Blockly extension, the self-explanation extension can be configured using query string parameters for `id`, `log`, and `se=1` (to enable the extension). The data is POSTed to the endpoint specified by `log` and consists of the contents of the code cell and the self-explanation. Pairing the code and self-explanation ensures that they are properly analyzed together as a snapshot in time, as the student is always free to rewrite the code, self-explanation, or both.

4. HTML INJECTION EXTENSION

The last extension we present, the HTML injection extension, is quite different from the others in that it does not intrinsically collect data. Rather, this extension is a template for creating extensions that can be used for various purposes, including collecting data. We developed this extension in response to a particular problem, displaying hosted videos embedded in JupyterLab. Typically this is done by executing Python and using associated widgets, but for experimental purposes we wanted to present embedded videos without associated code. Surprisingly this is not possible under the JupyterLab security model, which forbids JavaScript running in Markdown cells.

To circumvent this limitation, we use the metadata associated with the Markdown cell to specify arbitrary HTML and JavaScript, which we then inject into the Markdown cell outside the standard JupyterLab rendering of the notebook. The extension allows for easy embedding of video in the JupyterLab notebook, as shown in Figure 3. Any other rich media disallowed by JupyterLab's security model can be embedded in the same way, as can JavaScript that executes data collection functions. However, we note two caveats with this approach. First, the elements specified in the metadata will not render on servers that do not have this extension installed, making the notebooks that depend on it non-portable to some extent. Second, the extension allows for non-obvious code to be run when a notebook loads, so it is not recommended for use outside the research context.

5. CONCLUSION

We have presented three JupyterLab extensions to enable educational data mining research in CSEd and data science. Two of these, the Blockly extension and the self-explanation extension, support direct logging of data to a standard endpoint for POST requests. The Blockly extension provides clickstream-level data for blocks manipulation that can be used for future research on learning programming. The self-explanation extension provides rich natural language data reflecting student reasoning during problem solving. The third extension, the HTML injection extension, can be used flexibly for presentation of rich media or for data collection. All extensions are open-source and distributed through NPM at `https://www.npmjs.com/~aolney`. We hope these extensions will enable future work using JupyterLab as a source of student data for educational data mining.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants 1918751 and 1934745 by the Institute of Education Sciences under Grant R305A190448. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the Institute of Education Sciences.

7. REFERENCES

[1] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari. From Scratch to "real" programming. *ACM Transactions on Computing Education*, 14(4):25:1–25:15, Feb. 2015.

[2] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak. Learnable programming: Blocks and beyond. *Communications of the ACM*, 60(6):72–80, May 2017.

[3] M. T. H. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13:145–182, 1989.

[4] M. T. H. Chi, N. de Leeuw, M. H. Chiu, and C. LaVancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3):439–477, 1994.

[5] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 141–146, New York, NY, USA, 2012. ACM.

[6] Google. Blockly, 2019. original-date: 2013-10-25T21:13:33Z.

[7] J. B. Hamrick. Creating and grading IPython/Jupyter notebook assignments with NbGrader. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, SIGCSE '16, pages 242–242, New York, NY, USA, 2016. ACM.

[8] Kaggle. The state of ML and data science 2017, 2017.

[9] C. M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 346–350, New York, NY, USA, 2010. ACM.

[10] B. Moskal, D. Lurie, and S. Cooper. Evaluating the effectiveness of a new instructional approach. *ACM SIGCSE Bulletin*, 36(1):75–79, Mar. 2004.

[11] T. J. Nokes, R. G. M. Hausmann, K. VanLehn, and S. Gershman. Testing the instructional fit hypothesis: the case of self-explanation prompts. *Instructional Science*, 39(5):645–666, Sept. 2011.

[12] K. J. O'Hara, D. Blank, and J. Marshall. Computational notebooks for AI education. In I. Russell and W. Eberle, editors, *Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference*, pages 263–268. AAAI Press, 2015.

[13] A. M. Olney, S. D. Fleming, and J. C. Johnson. Learning data science with Blockly in JupyterLab. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1373, New York, NY, USA, 2021. Association for Computing Machinery.

[14] B. Peng, G. Wang, J. Ma, M. C. Leong, C. Wakefield, J. Melott, Y. Chiu, D. Du, and J. N. Weinstein. SoS notebook: an interactive multi-language data analysis environment. *Bioinformatics*, 34(21):3768–3770, 2018.

[15] 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*, ICER '15, pages 91–99, New York, NY, USA, 2015. ACM.

[16] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, Nov. 2009.

[17] A. Rule. We analyzed 1 million Jupyter notebooks – now you can too.

[18] H. Shen. Interactive notebooks: Sharing the code. *Nature News*, 515(7525):151, Nov. 2014.

[19] E. Tulving. How many memory systems are there? *American Psychologist*, 40(4):385–398, 1985.

[20] G. Wilson, F. Perez, and P. Norvig. Teaching computing with the IPython Notebook (abstract only). In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 740–740, New York, NY, USA, 2014. ACM.

APPENDIX
A. FIGURES

Additional figures that support the main text are shown below.

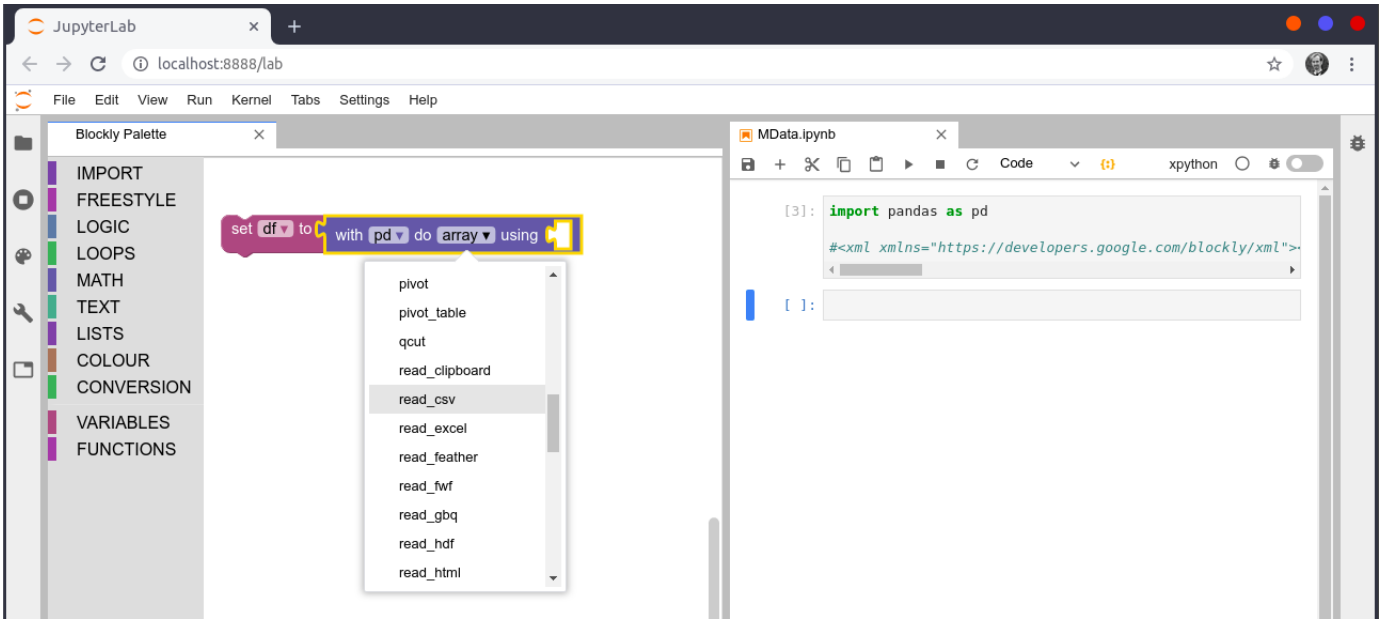

Figure 1: The `JupyterLab Blockly` extension showing the `Blockly` workspace on the left and notebook on the right. The workspace shows a dynamically-generated intelliblock for `pandas` with the possible function calls from the `pd` alias. Tooltips for `pd` and the functions not shown due to space limitations. The notebook on the right shows the code generated from a previous `import` block with the commented serialized XML used to regenerate that workspace when the cell is clicked again using notebook sync.

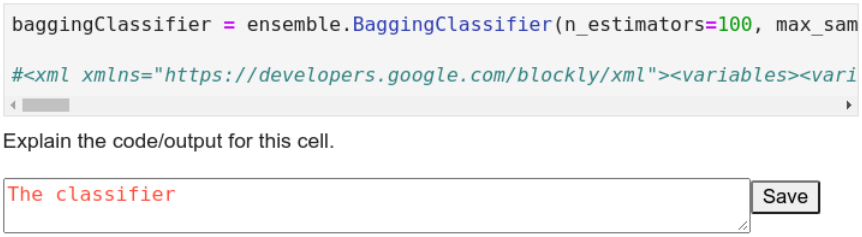

Figure 2: The `JupyterLab self-explanation` extension showing the text entry box it appends to the bottom of every code cell. As the student types, the text changes red to indicate unsaved changes. When the student presses the "save" button, the self-explanation is logged and the text changes to black.

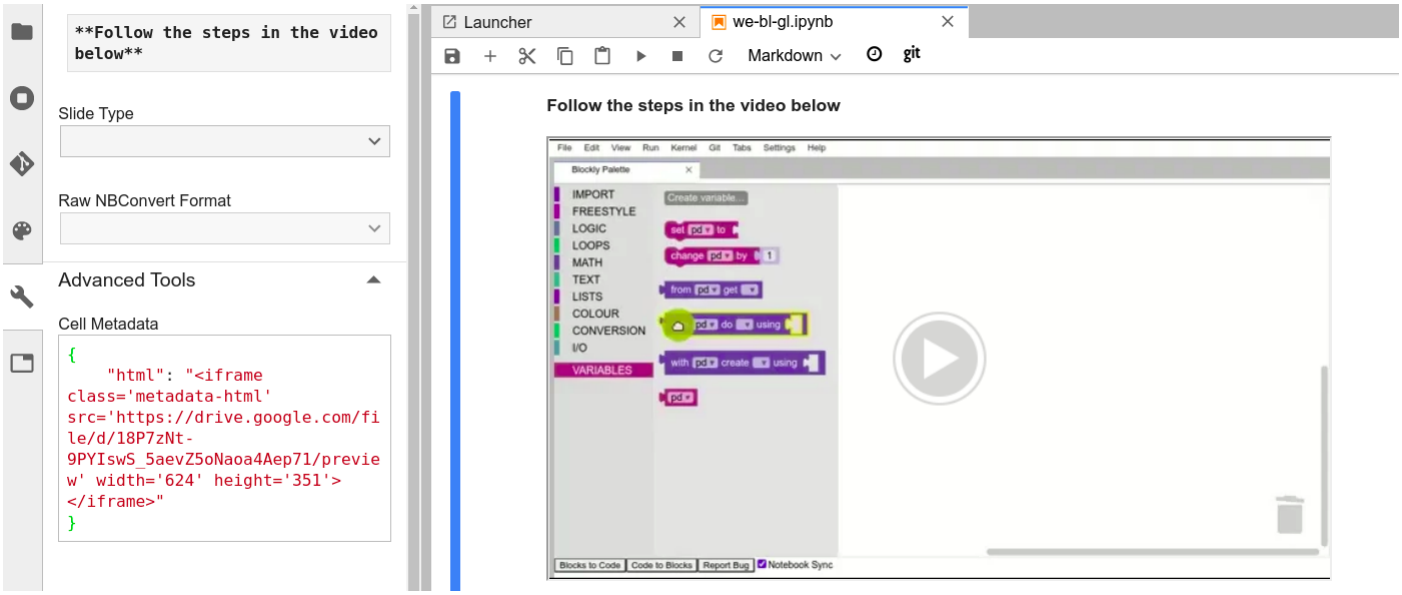

Figure 3: The `JupyterLab HTML injection` extension showing a video embedded directly in the notebook markdown via the metadata for that cell, circumventing the JupyterLab security model. Note that such elements will not render if the notebook is viewed on a server without this extension installed.