International Society of the Learning Sciences

Low-Friction Transition from Blocks to Python

Devin Jean, Brian Broll, Gordon Stein, Ákos Lédeczi
devin.c.jean@vanderbilt.edu, brian.broll@vanderbilt.edu, gordon.stein@vanderbilt.edu, akos.ledeczi@vanderbilt.edu
Vanderbilt University

Abstract: Block-based programming languages are quite successful in introducing students to programming through their intuitive interfaces. Tools like Scratch and Snap*!* empower students to create a variety of projects like games, stories, or multi-agent simulations. NetsBlox, a Snap*!* variant, even gives students user-friendly access to web-APIs and networked communication between projects and devices. Nevertheless, students eventually want to learn textual languages like Python. We introduce PyBlox, which aims to provide a smooth transition by keeping many of the familiar concepts from block-based environments, such as turtle graphics, multiple sprites, costumes, the stage, event-based programming, and the concurrency model, as well as access to web APIs and message passing through NetsBlox. This approach lets students focus on only the differences between the block-based language and Python itself. The paper discusses the fundamental concepts behind PyBlox, as well as how it can be used in practice to better teach Python to transitioning students.

Introduction

It is well-established that block-based coding interfaces can make it significantly easier to introduce K-12 students to programming while maintaining high interest (Broll et al., 2021; Weintrop & Wilensky, 2017). This is due to several factors, including the visual aspect of being able to move around blocks of code in 2D space, the rich collection of simple code snippets (blocks), and the inherent prevention of many classes of syntax errors due to constraints on block connections. Another key factor is that several block-based environments, such as Snap*!* and its extension, NetsBlox, empower students to write highly concurrent programs without even realizing the complexities that would be entailed in typical textual languages (Harvey & Mönig, 2010). This allows students to create advanced projects where multiple sprites simultaneously execute their scripts, with typically no synchronization burden placed on the student. Both environments enable students to easily create advanced behaviors with just a handful of blocks, which makes them intriguing to novice programmers. NetsBlox in particular can invoke interest in students due to its novel inclusion of network-related blocks for accessing web-based resources and distributed computing concepts like message passing between projects (Broll et al., 2018).

However, as students advance, the constraints of blocks can become detrimental, and many students want to explore textual languages instead. Unfortunately, typical textual programs are quite different from the convenient, graphical, concurrent environments these students would be familiar with, meaning there is limited direct knowledge transfer aside from computational thinking and basic control structures. This also runs the risk of students forgetting any advanced network-based CS topics they might have learned in NetsBlox, making students fall back to more simplistic projects than they might like or feel that they are not making progress if they cannot recreate these more advanced projects.

PyBlox aims to solve these issues by being a transitional tool between NetsBlox and Python. In PyBlox, students can use Python to create graphical, sprite-based projects in much the same way as in NetsBlox. The PyBlox IDE is designed to be familiar to students coming from block-based environments, with the same breakdown of code tabs for different sprites and a palette of blocks which can be dragged and dropped into the editor to paste a code snippet. PyBlox also mimics the concurrency model of NetsBlox, so direct translation from blocks to Python is possible with only minimal changes in behavior. Additionally, PyBlox exposes all the network-based utilities of NetsBlox through a nearly identical interface. With all of these affordances, students can transfer much more of their existing block-based project knowledge into Python, allowing them to focus on the core difference: the Python language itself.

PyBlox is certainly not the first project to attempt to bridge block-based and textual languages for the purpose of transferring skills (Dann et al., 2012; Wagner et al., 2013). A well-known tool is Pencil Code, which allows for the bidirectional conversion between blocks and two textual languages: Javascript and Coffeescript (Bau, 2015). This serves the initial requirement of bridging blocks and text, but in doing so, it brings complicating details into the block-based environment. For instance, although multiple turtles/sprites are allowed, they are all controlled by a single procedural script. Some event-based utilities are provided, so it is not entirely synchronous,

but they still execute globally and must instruct specific turtles to do specific things. This breaks the separation of sprites into self-driving agents with their own scripts, as NetsBlox and PyBlox have.

There are also other projects like EduBlocks which support block-to-text conversion for several textual languages including Python and HTML; however, the provided blocks are specific to the target textual language (EduBlocks, n.d.). In general, existing block-to-text transitional tools often tailor the provided blocks to the textual language in some way. However, to be useful as a transitional tool, students must be familiar with the provided blocks, so changing the blocks to suit the language is arguably counterproductive unless students are taught using that tool to begin with. PyBlox instead starts with a popular block-based environment like Snap!/NetsBlox and creates different but equivalent Python interfaces which support all of the necessary behaviors. Additionally, PyBlox does not restrict the textual language to a subset that can be translated back into blocks.

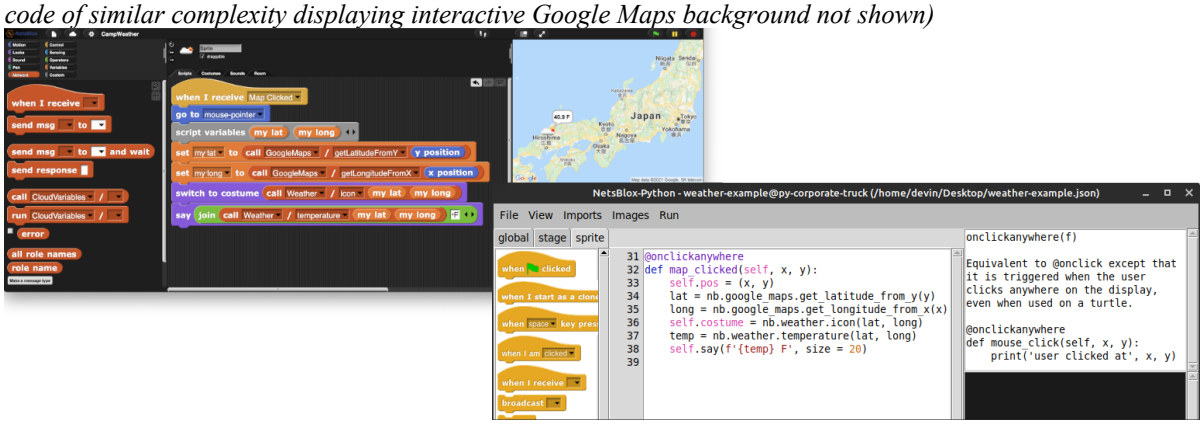
The most similar existing work is Pytch, which is a graphical, browser-based python environment (Pytch, n.d.). While Pytch and PyBlox are similar in philosophy and usage for the core turtle graphics features and event-based computing, PyBlox offers several benefits for users. For instance, Pytch does not break down code into tabs representing sprites (exposing users to boilerplate class definitions), does not support the networking features that PyBlox provides through NetsBlox, and is trapped in the browser, missing out on powerful native libraries. Meanwhile, PyBlox, in exchange for the burden of installation, gives students access to the full power of Python.

PyBlox IDE

The PyBlox IDE is designed to have a similar general layout as used by Snap*!* and NetsBlox. The code editor breaks down the project into different tabs representing the stage and various sprites, as well as a tab for anything that must be declared at global scope, such as global variables. Behind the scenes, tabs representing the stage and sprites generate Python class definitions, but this detail is hidden from students for simplicity. Thus, only script/method definitions need to be written, which makes the structure of the PyBlox program look quite similar to NetsBlox code and avoids the need to manually enter boilerplate code. Access to stage/sprite utilities can be done by changing properties or calling methods on the given object, which can easily be discovered with the built-in suggestion/completion system. Context sensitive documentation and examples of usage are always shown in the documentation panel in the top-right corner of the editor. The NetsBlox and PyBlox editors with equivalent projects loaded are shown in Figure 1.

Figure 1

NetsBlox and PyBlox with equivalent projects displaying current weather conditions at the mouse click. (Stage code of similar complexity displaying interactive Google Maps background not shown)



The main conceptual difference between NetsBlox and PyBlox is that NetsBlox is "lively," meaning the program is always running. This is a relatively common feature in block-based languages but would be extremely unusual in traditional textual languages. Once written, PyBlox code is run separately in another window; students can edit their code during execution, but it will not affect the running program, which must be restarted to apply changes. An example of a running project is shown in Figure 2.

Like NetsBlox, PyBlox includes a panel of blocks on the left side of the screen which can be dragged and dropped to paste a code snippet into the editor. This is intended to contain a subset of blocks whose PyBlox equivalent would not be obvious to users initially, such as hat blocks (see next section). The set of blocks present in a project can be customized and is saved in the project file. This enables teachers to create and distribute starter projects with custom collections of blocks for a given project or activity.

The “Imports” menu has several curated Python packages which can be toggled on/off to automatically import them into the project without needing any code. A description of each package and example use cases are shown in the documentation panel when hovered over. This lets students easily find useful and reputable packages. However, even non-curated packages can be imported by manually using import statements. The “Images” menu lets users manage or add imported images; these are accessible from a running project through the `images` object (e.g., `images.my_cat`), and are stored inside the project file for portability. This is used to facilitate custom costumes as present in NetsBlox. Costumes can be set by the `costume` property on a sprite or the stage (e.g., `self.costume = images.my_cat`).

Figure 2

Equivalent NetsBlox and PyBlox code (left) and the running PyBlox project (right)

Concurrency and Networking

One of the key features of block-based languages that makes them so powerful is their simple concurrency model, which allows many scripts on various sprites to run simultaneously. To do this, Snap*!* and NetsBlox use time-sharing where scripts are executed for up to some fixed duration of time or until a yield point is encountered (e.g., at each iteration of a loop, the current script gives up the CPU). Thus, only one script is actually running at a time, but scripts are being switched around rapidly creating the impression of simultaneous execution. Conveniently, Python itself uses time-sharing for its implementation of concurrency, so we can already mimic much of the basic NetsBlox concurrency model. To add the additional yield points (e.g., the ones inside loops), PyBlox projects are passed through a preprocessor before execution, which breaks the student’s code into a syntax tree and rebuilds it with the added yield points and other linking/boilerplate code.

Figure 3

Comparing NetsBlox and PyBlox events. Hat blocks are the first blocks of event handler scripts.

Specifically, the concurrent properties of Snap*!* and NetsBlox scripts come from “hat” blocks which go on top of a script and act like event handlers. To facilitate this same effect in Python, several simple decorators are provided; for instance, the `@onstart()` function decorator is directly equivalent to the “When green flag clicked” hat block. The presence of a decorator in PyBlox turns a Python “function” into a “script” from the students’ perspective and allows it to run concurrently. Behind the scenes, PyBlox manages the concurrent execution of these scripts, including the management of execution queues for event-like scripts such as “When I receive” blocks for receiving messages. Figure 3 includes examples of some of the supported decorators compared to their block-based counterparts.

PyBlox is not just a Python-based tool for transitioning students from blocks to textual languages: it also brings with it simple ways to access all existing NetsBlox resources. This includes all supported online services, which execute on the NetsBlox server to access and return data from web APIs, as well as message passing

between projects. The PyBlox interfaces for these features are also very similar to NetsBlox, as shown in Figures 2 and 4. These PyBlox wrappers are programmatically generated from NetsBlox service metadata, allowing PyBlox to stay up to date with future services as NetsBlox continues to expand. Note that these generated wrappers include documentation which can be seen in the documentation panel of the IDE.

Figure 4

Comparing NetsBlox and PyBlox web service access and message sending (for receiving, see Figure 3)


```
nb.air_quality.quality_index(lat, long)
```


```
nb.send_message('message', target, msg='hello')
```

PyBlox in Education

As already seen, PyBlox supports most of the commonly used Snap*!*/NetsBlox features in ways that are visually quite similar to their block-based counterparts. However, some differences (e.g., properties) are included to be more "Pythonic" and provide students gentle exposure to more advanced textual language features and avoid variable getters/setters. Because many students are already being introduced to programming with Snap*!* or its derivatives, PyBlox can be used as a drop-in transitional tool for students without having to use a specialized block-based environment meant to be used as/with a transitional tool. From a student perspective, the only initial problem is knowing how to invoke certain behaviors like "move _ steps." However, this can be solved by the blocks palette shown in the PyBlox IDE, which can be configured on a per-project basis by an instructor to show the blocks that students might need, or even completely custom ones designed by the instructor.

Additionally, PyBlox provides access to all existing NetsBlox services and message passing, which have previously been used to introduce students to advanced network-related CS topics like cybersecurity, robotics, distributed computing, and the internet of things (Lédeczi et al., 2019). With PyBlox, these advanced topics and projects can extend directly into Python for continued learning. Because PyBlox can also be imported as a normal Python module (called `netsblox`) to access NetsBlox utilities without the opt-in graphical sprite environment or IDE, instructors could even include these advanced CS topics in their existing Python-based curriculum with very little overhead since students would already be familiar with Python.

Overall, transitioning students will be able to keep much of their block-based project knowledge, including the ability to create powerful, concurrent, and networked projects, while absorbing Python syntax along the way. Currently, we are planning studies with high school students to evaluate PyBlox and examine transfer learning from NetsBlox to PyBlox/Python.

References

Bau, D., Bau D. A., et al. (2015). Pencil Code: Block Code for a Text World. *Proceedings of the 14th International Conference on Interaction Design and Children*, 445–448.

Broll, B., Lédeczi, Á., Stein, G., Jean, D., Brady, C., Grover, S., Catete, V., & Barnes, T. (2021). Removing the Walls Around Visual Educational Programming Environments. *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 1–9.

Broll, B., Lédeczi, Á., Zare, H., Do, D. N., Sallai, J., Völgyesi, P., Maróti, M., Brown, L., & Vanags, C. (2018). A Visual Programming Environment for Introducing Distributed Computing to Secondary Education. *Journal of Parallel and Distributed Computing*, 118, 189–200.

EduBlocks (n.d.). Retrieved December 2, 2021, from https://edublocks.org/.

Harvey, B., & Mönig, J. (2010). Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? *Proceedings of Constructionism*, 1–10.

Lédeczi, Á., Maróti, M., Zare, H., Yett, B., Hutchins, N., Broll, B., Völgyesi, P., Smith, M. B., Darrah, T., Metelko, M., Koutsoukos, X., & Biswas, G. (2019). Teaching Cybersecurity with Networked Robots. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 885–891.

Pytch (n.d.). Retrieved March 7, 2022, from https://www.pytch.org/app/.

Wagner, A., Gray, J., Corley, J., & Wolber, D. (2013). Using App Inventor in a K-12 summer camp. *In Proceedings of the 44th ACM technical symposium on Computer Science Education*, 621–626.

Weintrop, D., & Wilensky, U. (2017). Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions Computing Education*, 18(1), 1–25.

