–Independent Work Report Spring,2014–

Python Pieces: An Educational Programming Environment

April 30, 2015

Matthew Wang ’16, advised by Dr. Christopher Moretti
Undergraduate, Department of Computer Science, Princeton University

Abstract

As computing technology continues to transform every aspect of life in the 21st century, learning about computer science is becoming increasingly important. But most students in the U.S. first study computer science in a college course, if ever. Most existing educational tools are either syntax focused tutorials of programming languages for high schoolers and adults or kid-friendly visual environments that are fun to program but yield few transferable programming skills. Python Pieces aims to provide a bridge between these two camps and provide a beginner oriented programming environment for a real programming language, suitable for a middle school student. The student experience centers around using draggable Code Blocks to create Python programs. Teachers can use Python Pieces to prepare lessons and even integrate student programs with external code.

Contents

1. Introduction

As computing technology continues to transform every aspect of life in the 21st century, learning about computer science is becoming increasingly important for everyone, everywhere. But there is huge disparity between the demand for computer science skills in the job market and the priority K-12 schools place on teaching students such skills. As of 2010, only 14 out of 50 states in the United States have incorporated computer science as a significant part of their secondary school curricula, and no states require any kind of computer science course to graduate high school [12]. As a result, most students in the U.S. first study computer science in a college course, if ever.

Recently, a number of educational tools and services have arisen to address this lack of computer science instruction for younger students. There are numerous online courses and tutorials, such as Codecademy [6] and Coursera [3], that are meant to facilitate self-teaching of various programming languages. These courses usually introduce increasingly advanced parts of a given language's syntax, interspersed with simple coding exercises to improve understanding and occasional projects, typically with some semblance of applicability in real life, that require synthesis of new material with prior content. The student writes the code in some kind of text editor, and usually receives feedback on their program from a console's textual output.

However, traditional text-editor and console programming environments can put off the young beginner. Much of the instruction is focused on syntax rules that must be carefully observed to avoid errors, which can be intimidating and frustrating for beginners. Furthermore, mere textual feedback from a console may seem insufficiently rewarding for the hard work of writing conceptually sound and bug-free code. These sorts of courses and tutorials are better suited for high school to adult aged learners.

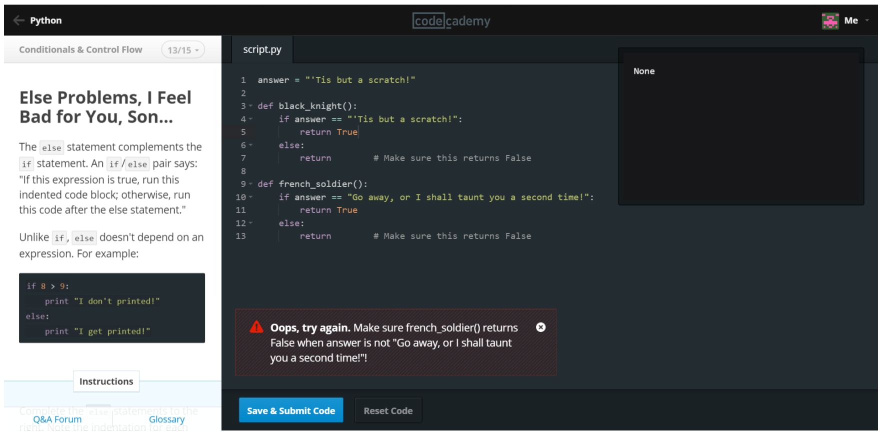
There are also alternative computer science education programs like Scratch [9], Kodu [2], and App Inventor [10] that aim to be more appealing to younger students than a text editor and console. These eschew well known programming languages in favor of more readable kinds of pseudocode. Users create programs by sequencing blocks of pseudocode selected from a list of choices, rather than typing them out, and can then visualize their program's execution in some kind of exciting virtual world. By providing preset blocks, students worry less about correct syntax and are able to freely explore the creative possibilities available to them in their virtual world. They can also be exposed to certain high level concepts such as object-oriented programming sooner. However, while these programs are easier to start working with and often more gratifying to run, there is a limit to what can be taught or learned in these environments because of the finite limits of pseudocode and the virtual world. A major criticism of these programs is that most of the skills learned are not transferable.

Python Pieces aims to provide a bridge between these two camps of resources and provide a beginner oriented programming environment for a real programming language, suitable for middle school students. Python Pieces's core feature is the concept of the Code Block: for a given project, there is a finite set of Code Blocks provided that can be arranged via drag-and-drop to form a program. The Code Blocks contain actual Python syntax, not pseudocode, that when assembled read as Python script. Manipulating a limited set of Blocks is far less intimidating and error-prone than writing code in an editor, and should encourage more exploratory learning. But ultimately users are still working with a real programming language, so the skills they learn can be smoothly translated to a more traditional programming environment.

2. Existing Educational Environments: Case Studies

We will take a closer look at three educational programming environments – Codecademy, Scratch, and Pyland – and some of their highlights that inspired more design principles of Python Pieces.

2.1. Codecademy



Else Problems, I Feel Bad for You, Son...

The `else` statement complements the `if` statement. An `if` / `else` pair says: "If this expression is true, run this indented code block; otherwise, run this code after the else statement."

Unlike `if`, `else` doesn't depend on an expression. For example:

```
if 8 > 9:
    print "I don't printed!"
else:
    print "I get printed!"
```

```
answer = "'Tis but a scratch!"

def black_knight():
    if answer == "'Tis but a scratch!":
        return True
    else:
        return        # Make sure this returns False

def french_soldier():
    if answer == "Go away, or I shall taunt you a second time!":
        return True
    else:
        return        # Make sure this returns False
```

```
None
```

Oops, try again. Make sure french_soldier() returns False when answer is not "Go away, or I shall taunt you a second time!"!

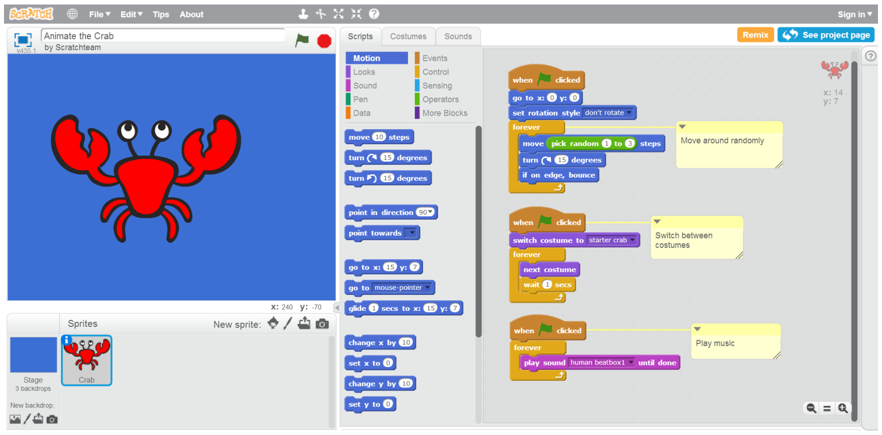
Figure 1: A Python lesson screen on Codecademy [6]

Codecademy [6] offers comprehensive beginner courses in HTML and CSS, Javascript, jQuery, Python, Ruby, PHP, and more using a comprehensive, in-browser programming environment. Courses are broken into topical units, and each unit contains multiple lessons. Each lesson screen has a large text editor, a panel on the left with the lesson instructions, an emulated interactive console, and a large "Save & Submit Code" button that runs the script in the console, evaluates whether the code is correct, and triggers popups with helpful hints if it does not. Upon successful submission of one lesson's code, the user is prompted to proceed to the next lesson. User accounts track progress through courses.

Codecademy is currently the first Google search result for "learn to code," which is some indication of how well it works. In particular, Codecademy's pace of instruction from lesson to lesson is steady but not overwhelming, introducing new concepts gradually and constantly reviewing old ones. The text editor has good syntax coloring that helps understanding, and its interactive console allows for rapid testing and editing cycles. Lastly, the big "Save and Submit" button that saves, runs, and evaluates the student's code is clear, friendly looking, and provides helpful feedback, encouraging the user not to be afraid of running their code and engaging in frequent test and edit cycles.

2.2. Scratch

Scratch is a "creative learning community" developed by the Lifelong Kindergarten Group at the MIT Media Lab, where "you can program your own interactive stories, games, and animations" [9]. Programs are built from a large set of draggable puzzle-piece shaped units of pseudocode called blocks, which describe different behaviours of various visual objects in a 2-dimensional environment. These blocks may have special ways of connecting to other blocks, such as a loop that contains an arbitrary number of other blocks, or adjustable fields, like the number of times that loop should execute. Users can save their creations and share them with the online community, as



```
when [flag] clicked
go to x: 0 y: 0
set rotation style don't rotate
forever
    move pick random 1 to 3 steps
    turn 15 degrees
    if on edge, bounce

when [flag] clicked
switch costume to starter crab
forever
    next costume
    wait 1 secs

when [flag] clicked
forever
    play sound human beatbox1 until done
```

Move around randomly

Switch between costumes

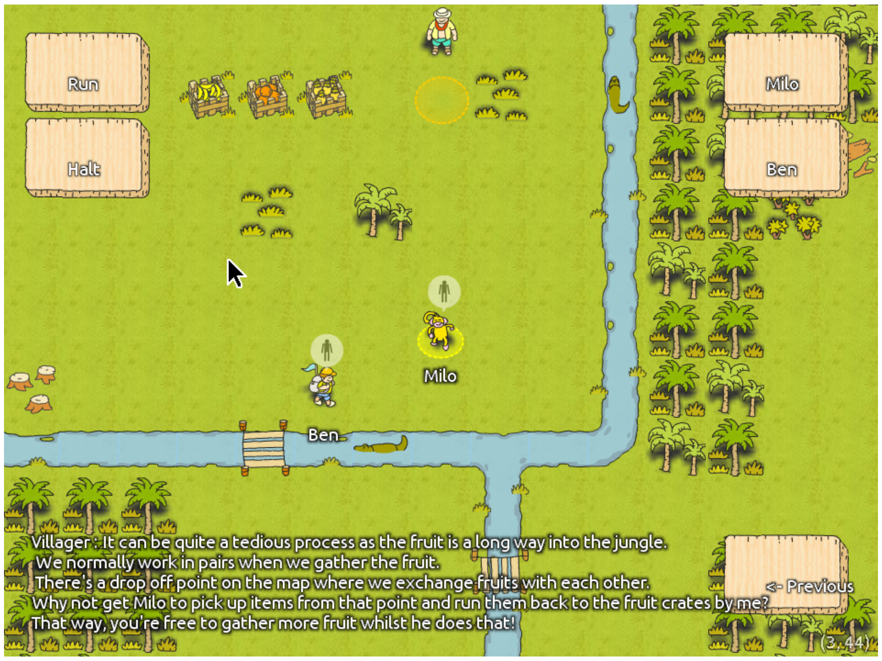
Play music

Figure 2: The Scratch programming environment [9]

well as “remix” other users’ shared projects, creating a copy of the program and tweaking it to their heart’s content.

Scratch’s interlocking blocks are the cornerstone of its programming experience, and dragging and dropping them between the left side container and the right side program area is intuitive and satisfying. The different colors of Scratch’s blocks are visual reminders of function, and the panel on the left contains all the blocks you might use. Lastly, the constant availability of feedback that is visual and interactive in nature makes testing and editing enjoyable rather than frustration or scary, which encourages frequent test and edit cycles.

2.3. Pyland



Villager : It can be quite a tedious process as the fruit is a long way into the jungle.
We normally work in pairs when we gather the fruit.
There's a drop off point on the map where we exchange fruits with each other.
Why not get Milo to pick up items from that point and run them back to the fruit crates by me?
That way, you're free to gather more fruit whilst he does that!

Figure 3: A screen from the Pyland game [5]

Pyland is an ongoing project from the University of Cambridge Computer Lab that “aims to provide a fun and creative environment...to aid children learning programming and Computer

Science Concepts" [5]. It is a 2-dimensional game in which the main characters can be controlled by Python code written in an integrated text editor by the player. It is intended to fill the educational gap between Scratch and Python, targeting children of age 11-12 [4], an area in which Pyland is an innovator. Users are learning to program in Python, but in the context of an exciting visual, interactive environment. Though the game is limited in scope, the programming skills learned in Pyland can be translated into other programming contexts readily.

3. Why Python?

Python is a relatively simple yet expressive programming language, and thus well suited for Python Pieces' target audience of middle school aged beginners. The Python website's educational page offers the following:

> "Python offers an interactive environment in which to explore procedural, functional and object oriented approaches to problem solving. Its high level data structures and clear syntax make it an ideal first language, while the large number of existing libraries make it suitable to tackle almost any programming tasks." [11]

As a "scripting" or "interactive" language, Python requires no compilation and is interpreted at run time. This eliminates compile time errors that can frustrate the edit-run-debug cycle [13, 8], as well as foster the illusion that successful compilation is equated with a correct program [13]. The built-in high level structures, including tuples, lists, and dictionaries, are powerful enough to handle a wide range of problems and yet simple in syntax and usage [8]. Python's vast and ever-growing number of libraries allow beginners to get beyond toy programs quickly and create more substantial programs faster.

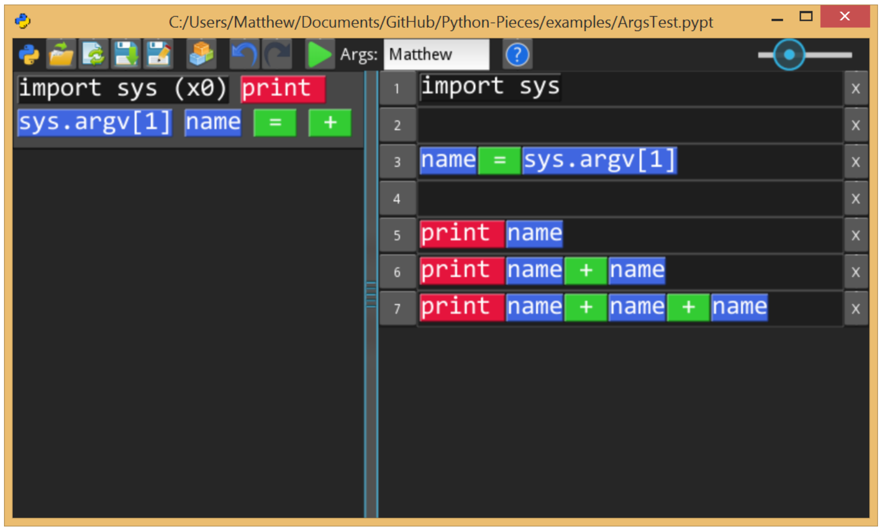
Python's dynamic typing is another advantage: less space, time, and errors are wasted with variable type declarations, and programmers can design their code's logic "on their feet" rather than having to plan in advance about the variables they may use [8, 13, 7]. Yet another selling point of Python is the simple and ubiquitous "print" keyword/function that handles multiple data types and makes basic debugging in Python easy [8]. Furthermore, Python's minimalistic and readable syntax make it a good candidate for being represented in "blocks."

Python is also a solid choice from a practical standpoint. The language comes preinstalled on most modern UNIX systems, and installing Python is minimal work. Python is completely safe from memory leaks and segmentation faults. Python is also becoming popular and respected in software industry, so learning to program in Python yields marketable career skills immediately.

4. Python Pieces

Python Pieces aims to integrate some of the best features of Codecademy and Scratch help make learning Python easier and more enjoyable for the same adolescent audience targeted by Pyland. Similar to Scratch, Python Pieces uses Code Blocks that can be arranged via drag-and-drop to form a program. But since the Code Blocks represent bits of actual Python, not pseudocode, Python Pieces can be used to teach Python in lesson form similar to Codecademy. Finally, Python Pieces can run the students' code both on its own or integrated with external code to provide immediate feedback, from basic console outputs to rich visualizations of the program.

Python Pieces's interface consists of one main screen, divided into two resizable halves. On the left half of the screen is the Block Box, where all the available Code Blocks can be accessed. On


C:/Users/Matthew/Documents/GitHub/Python-Pieces/examples/ArgsTest.pypt

Args: Matthew

```
import sys (x0) print
sys.argv[1] name = +
```

```
import sys

name = sys.argv[1]

print name
print name + name
print name + name + name
```

Figure 4: The Python Pieces interface

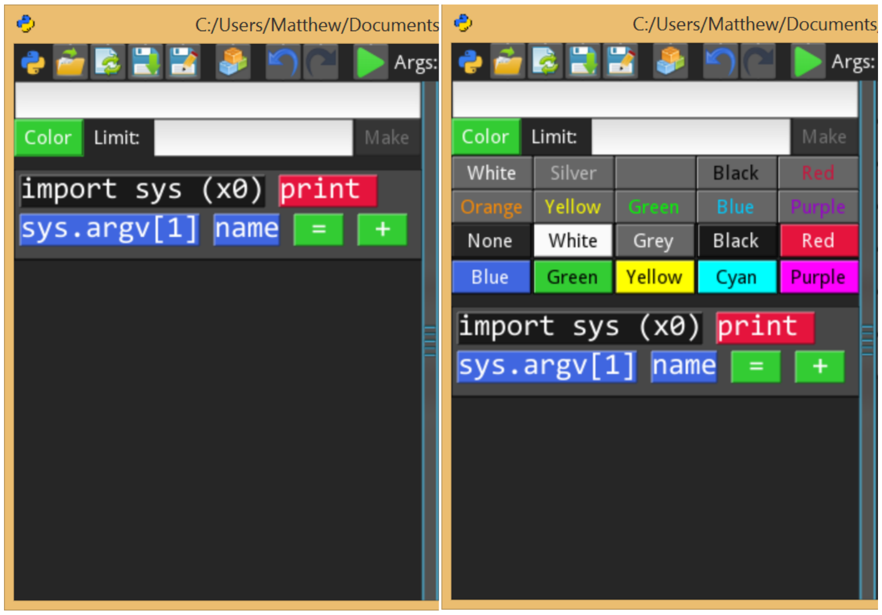
the right half of the screen is the Code Space, which contains a number of Code Lines. The student can drag Code Blocks from the Block Box to the Code Space, where they are dropped into the Code Lines to form the program. The student can also remove and rearrange Code Blocks in the Code Space by dragging. This resembles the Scratch environment, where all usable Blocks can be found on the left, and the program is assembled on the right. When the student clicks the Run Code button, the Code Space is translated into a Python script and run in a new console, giving immediate feedback on how the program works, similar to Codecademy's "Save and Submit" button.

Python Pieces is designed to help teachers provide incremental lessons for their students at a pacing similar to Codecademy's Python course. For a given lesson, the teacher can prepare a project template for their students, using the Code Block Maker to choose Code Blocks that are initially available to use and placing preliminary Blocks in the Code Space. These Code Blocks can be given text and background colors as visual reminders of their function, similar to Codecademy's syntax coloring and Scratch's block colors. The teacher can even specify how many times a particular Block may be used in the program in order to discourage or encourage particular approaches to constructing the program. The student can start his project from the template and see what Blocks he is expected to work with, and can then manipulate them freely as he works towards a complete program. The teacher can then reuse templates as a basis for future lesson templates, adding a few blocks for new content and building upon the content from previous lessons.

Python Pieces also provides a way for teachers to integrate the code written by the student using Blocks with external code; this functionality greatly enhances the kind of feedback the student can receive. The teacher could, for example, prepare some kind of auto-grader for students' code similar to Codecademy, or incorporate the student's code into a visual environment like Scratch or a game like Pyland.

5. Major Design Decisions

It became clear soon after conceptualizing the idea of Code Lines to contain the Code Blocks that Code Lines would need functionality similar to Code Blocks, including the ability to reorder, add, and remove Lines. The first implementation was a set of buttons for each Code Line: move up, move down, and add-new-line-after buttons on the left, and a delete the line button on the right.



```
import sys (x0) print
sys.argv[1] name = +
```

Figure 5: The Block Maker toggled onscreen, with color picking toggled on and off

This turned out to make the left side very cluttered looking and difficult to avoid pressing the wrong button by accident. Furthermore, moving a Code Line farther than one line up or down required repeatedly pressing a button that was moving after every press. The solution to the second problem was to use a drag-and-release interaction to reorder Code Lines; this was faster and also more consistent with the intuitive nature of the Code Block dragging. This collapsed the two move-up and move-down buttons into one drag-to-move button, partially solving the first problem of being too cluttered. The drag-to-move button was then combined with the line number label, and the line adding functionality was mapped to a double click on that button. This left only one button on each side, eliminating the clutter.

Another important question that arose early in design of Python Pieces was if the adding and removal of available Code Blocks in the Block Box should be restricted to teacher usage. The eventual decision was to let the Code Block Maker be available to both students and teachers, but allow it to be toggled on and off the screen. The default position is off, consistent with the idea that the student should primarily work with what is already provided by the teacher. A teacher may then see fit to forbid or limit the student's use of the Code Block Maker for a particular lesson, and since all Code Blocks used in the program must also be in the Block Box, a student could not use custom Code Blocks in his program without his Block Box clearly showing that he did so.

Yet another major design decision for Python Pieces was whether to predefine a set of common Python structures as defaultly available Blocks, similar to how Scratch users start with a large set of usable blocks. Examples would include an indenting tab, common operators like '=' or '+', keywords like 'if' and 'print', and popular built-in functions like 'int()' and 'str()'. The final choice was to not provide such a set; this shifts the main responsibility over to the teacher to determine what blocks the student may use in any given project, as well as define what the colors of different blocks should mean. This choice comes at the cost of making open-ended exploration with Python Pieces daunting, but it is an acceptable cost since the educational approach is closer to Codecademy's guided instruction than Scratch's open playground. Since Python is a large language with many popular constructs, any attempts at defining such a set would grow impractically large and look overwhelming to novices but still probably leave out parts of Python that one might expect to be part of the core set.

To handle template functionality, Python Pieces uses two different file extensions: .pyp (PYthon Pieces project) for student projects, and .pypt (PYthon Pieces project Template) for teacher-made templates. This is enforced by convention only, as the two kinds of files are represented the same way, and either can be loaded from or saved to freely. However, the Save button saves to a .pyp file by default, even if the project was originally loaded from a .pypt file, so as not to overwrite the template by accident. The Reset To Template button simply looks in the directory of the project for a .pypt file of the same name and reloads the workspace from that file. This separation of the two kinds of files is simple to work with loading and saving, compared to the alternative solution of embedding the original template information within a project file, but it hinders the student's ability to save his project under a different name and have different versions of his work since Reset To Template requires the file name to be the same as the template. This issue is discussed further in the Future Work section.

How student code could be integrated with external code was another major design decision. In the final product, a teacher can (optionally) prepare external Python code in a file of the same name as the template and a .pypc file extension (PYthon Pieces Code template). When the student clicks the Run Code button, the resulting Python script that is created and run will be the code in the .pypc file, but with the project code inserted wherever an "#INSERTHERE#" tag is found in the file, at the indicated indentation. This is a very "quick and dirty" solution, which is easy for a teacher to prepare and allows student code unlimited interaction with external code, but is potentially very unsafe since Python does not have private variables, and student code could cause unexpected side effects. One alternate solution was to always encase student code within some function in a different .py script than the main script to be run, limiting the student code's access to global variables in the external code. That solution, and others that involved creating separate scripts, ran into the issue of how to choose a file name for either the external code or the student code and still ensure linkage. Ultimately, it is again the teacher's responsibility to design his lesson well and safeguard against unsafe student code if he sees fit. This issue is also discussed further in the Future Work section.

6. Program Structure and Implementation Details

Python Pieces is created primarily with Kivy, which is an "Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps"[1]. Kivy was chosen as the main library because it gracefully supports popular GUI interactions such as swiping, scrolling, window-resizing, and dragging. Kivy is also cross-platform, so in addition to the current Windows build, MacOS and Linux versions of Python Pieces can be built with ease.

6.1. The Widget Tree and Layouts

Kivy applications are structured as a tree of "Widget" objects. The application is the root, and each Widget can have up to one parent Widget and zero or more child Widgets. Widgets are drawn to the screen in tree-traversal pre-order. Each Widget can have various properties and events which can be bound to function calls that will fire when the properties are altered or events are triggered. Many Widgets are various types of Layouts, which dynamically size and position their child widgets relative to themselves, or their own sizes and positions relative to their child widgets. The order of a Layout's child Widgets determines how the child Widgets may be represented in the Layout; for example, the Code Space Layout draws its Code Line+ children in order from bottom to top.

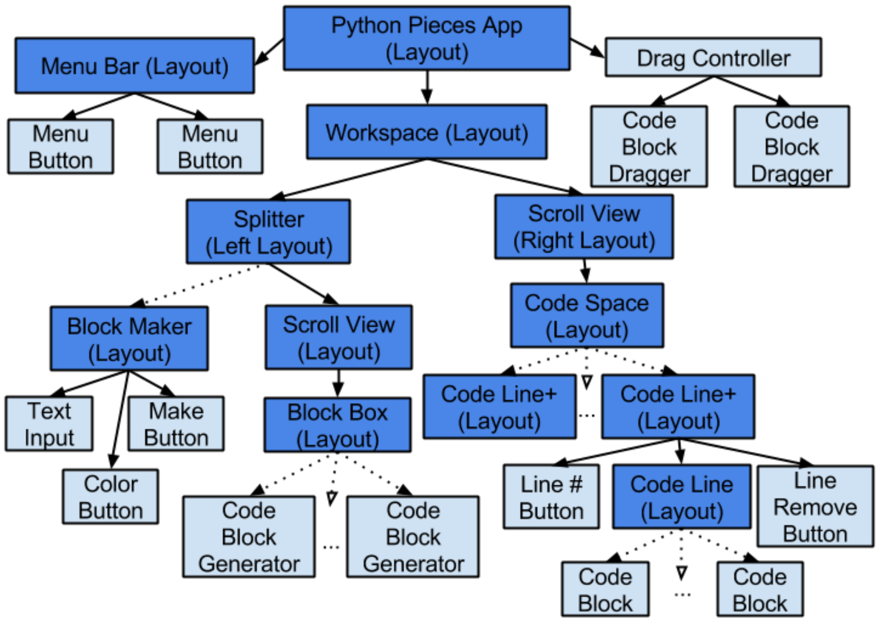


Figure 6: A representation of Python Pieces' main Widget tree

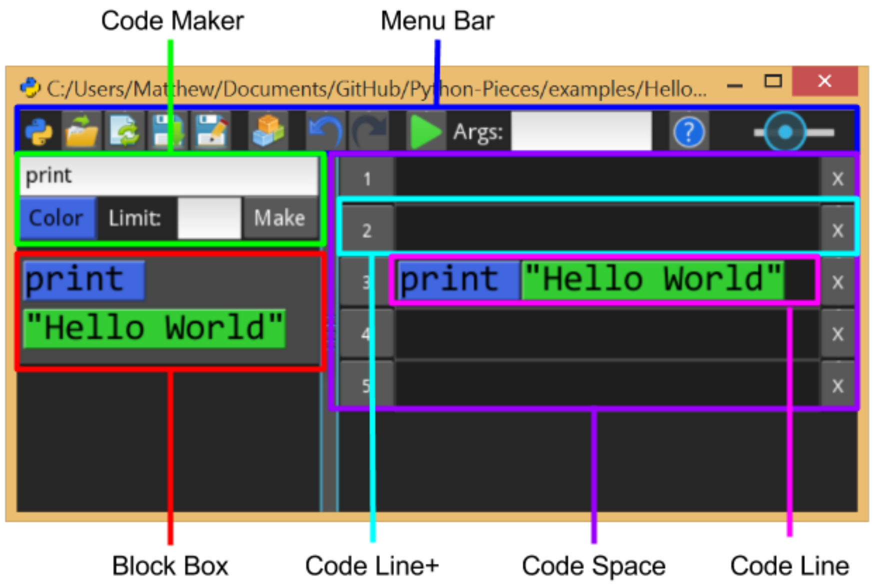


Figure 7: A Python lesson screen on codecademy.com

A somewhat streamlined representation of the main Widget tree for Python Pieces is drawn in Figure 6, and Figure 7 depicts the Python Pieces application screen with the main Layouts labeled. Parent-child relationships are represented by arrows, with the head pointing to the child: solid arrows indicate permanent relationships, while dotted arrows indicate relationships that may be severed or added during runtime. Note that names of the Widgets are similar to the terms used to describe the various parts of the UI earlier, but do not exactly match the user's perspective. To the user, Code Block refers to objects in either the Block Box or the Code Space, but a Code Block Widget only resides in the Code Space as a child of some Code Line, while the Block Box contains only Code Block Generator Widgets. Also, the Code Line Widget is the Layout where the Code Blocks reside in, while the Code Line+ Layout includes the two buttons on both sides of the Code Line as well as the Code Line itself.

The Menu Bar remains a fixed height at the top of the window, while the Workspace fills the remainder. The Workspace is divided into two halves via a vertical Splitter, which allows the user to resize the left half horizontally while the right half automatically grows or shrinks to fill the right half. In the left half, the Block Maker uses a fixed height if it is present, while the Block Box takes up as much vertical space below as necessary to hold all of its child Code Block Generators; if it gets too large for the window, vertical scrolling is enabled. The Code Space contains a series of Code Line+ Layouts, each of which contain a couple of buttons and a Code Line Layout that resizes horizontally to maximum width of the right half of the Workspace, and vertically to the height required by its contained Code Blocks (similar to the Block Space). If there are too many Code Line+ Widgets in the Code Space to be displayed in the window, vertical scrolling will be enabled (also similar to the Block Space behavior). Not mentioned in the widget tree is the Font Size Slider in the upper right hand corner (a child Widget of the App), which when adjusted will resize all Code Blocks, Code Block Generators, Code Line, and Code Line+, as well as the Block Box, and Code Space widgets that contain them accordingly.

6.2. Code Block, Code Generator, and Code Line+ Widgets

Python Pieces's interface centers around the user's manipulation of Code Blocks and Code Lines, and those manipulations affect the structure of the main Widget tree by adding, removing, and reinserting Code Block. To the user, the Code Generators in the Block Box and the Code Blocks in the Code Space are the same, but they behave differently. When touched or clicked, both Widgets create a Code Block Dragger that looks like a semi-transparent shadow of the Widget. The Code Block Dragger is added to the main tree as a child of the Drag Controller which causes it to be drawn in the foreground, and its position follows the mouse or touch until released. Upon release, the Code Block Dragger is removed from the tree, and depending on the source of the Dragger location of the click or touch release, action may be taken.

If the Dragger came from a clicked Code Generator, a release over a Code Line will create a new Code Block linked to the Code Generator as its source and add it as a child of the Code Line (unless the Generator has a usage limit that has been reached). A release within the Block Box will remove and reinsert the Code Generator as a child of the Block Box to reorder the Generators in the layout appropriately. If the Block Maker is on screen, a release over the Block Maker will remove the Code Generator from the Block Box's children, remove all Code Blocks that cite the Code Generator as its source from their Code Lines, and reset the Block Maker's state to the properties of the Code Generator.

If the Dragger came from a clicked Code Block, a release over a Code Line will remove the source Code Block from its Code Line parent and re-add it as a child of the new Code Line; a release over any other location will simply remove the Code Block from its Code Line parent, and notify its source Generator of its removal (to keep track of usage limits).

Similar to the Code Blocks, when the Code Line+ Widgets are dragged within the Code Space by the line number for reordering, it is removed from the Code Space's children and reinserted with the appropriate index. A double click on the line number adds a new empty Code Line+ Widget to the children of Code Space at the next index, and a click on the 'X' button removes the Code Line+ from the Code Space's children, as well as signals to the Code Generators of any associated Code Blocks that were removed with the Code Line.

Related is the Font Size Slider Widget, which has an internal font size value that changes when the user moves the slider. When that value is changed, the Widget tree is traversed to change the font size Property of each Code Line+, Code Block, and Code Block Generator, resizing each.

6.3. Version system and JSON file formats

Underlying most of the Menu Buttons (with the exception of Toggle Block Maker, Run Code, and Help buttons) is the Version system. To completely represent a state of the Workspace, there is a special Version object. Each Version object contains an ordered list of Generator Representation objects that represent each Code Block Generator in the Block Box. A Generator Representation includes the text, count (if applicable), text color, and background color of the Code Block Generator. Each Version also contains a list representing the Code Space, which contains lists that represent each Code Line. Each Code Line list contains a series of indices that represent each Code Block in that Code Line; each index refers to the index of the Generator Rep for the Code Block Generator that the Code Block is associated with.

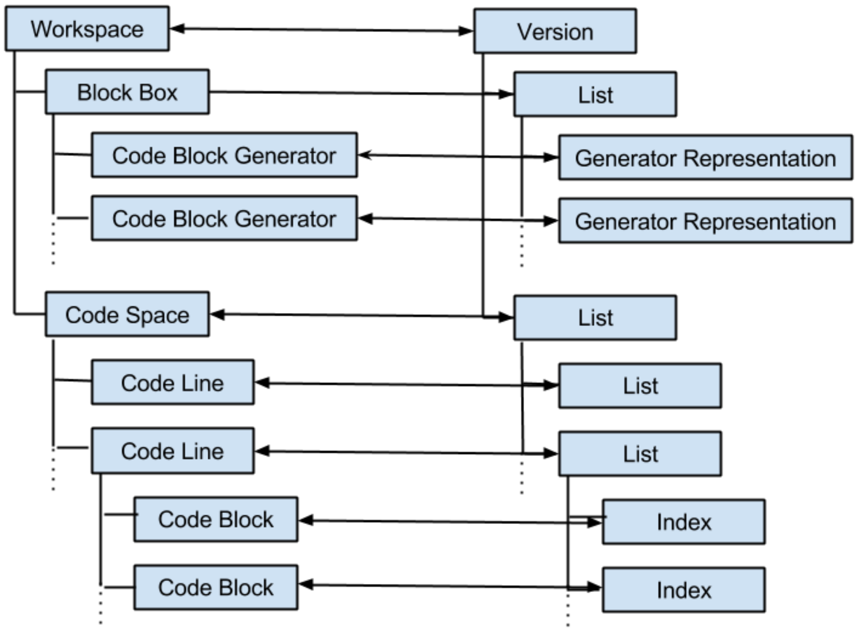


Figure 8: The Version object structure and how it maps to the Workspace state

From this bijective structure, as depicted in Figure 8, it is easy to create a Version from the Workspace by traversing the Widget tree and making the appropriate objects. One can also rebuild the Workspace from a Version: the Block Box is populated with Code Block Generators first, and then each Code Line populated with Code Blocks, linking each to their source Generator.

To handle Redo and Undo operations, a Redo and Undo stack of Version objects are kept and updated, as well as a current Version. After every workspace interaction that alters the contents of either the Block Box or Code Space, a new current Version is created and the old current Version is pushed onto the Undo stack. These interactions include every creation, deletion, or relocation (that is, changing child index and/or parent Widget) of a Code Block, Code Block Generator, or Code Line+ Widget. An Undo action pushes the current Version onto the Redo stack, then pops the most recent Version from the Undo stack as the new current Version and rebuilds the Workspace from it. Similarly, a Redo action pushes the current Version onto the Undo stack, then pops the most recent Version from the Redo stack as the new current Version and rebuilds the Workspace from it.

Maintaining stacks of entire states of the Workspace, as opposed to records of individual changes made to the Workspace, is relatively memory and performance inefficient. Making and rebuilding from Versions is far less bug prone, however, than recording and re-performing individual changes in the Workspace with some new data structure. Since the expected size of most projects is not large memory issues are not anticipated, but there is a maximum stack size of 100 Versions to avoid overburdening memory.

Both the .pyp and .pypt files, which are identical in all but extension, use a JSON file format to represent a Version object. JSON was chosen because the powerful, reliable Python library designed for it, and its readability which made it easier to debug. The readability is is both an upside and a downside; one could directly edit the JSON file without much trouble, which might be useful but is not very secure.

The Save and Save As operations take the current Version and represent it in textual JSON file format. The Open operation attempts to parse a file that is supposedly JSON and create a Version object, which if successful is used to build the Workspace. The Open operation also clears the Redo and Undo stacks, since the user should not be able to invoke changes from other files. The Reset To Template operation does the same as Open with a same-named .pypt file (if present in the same directory), but notably does not clear the Undo stack. This is because it is intended as a "start from scratch" option for students, but without permanently clearing out all records of previous attempts. Furthermore, it is a one-click operation that may be accidentally pressed.

6.4. Run Code

When the Run Code button is clicked, each Code Line is visited and of its Code Blocks are appended to form a string of a single line of code. The directory is then checked for a .pypc file of the same name as the project; if found, the text of that file is read into a string, and searched for instances of the tag "#INSERTHERE#." For each instance, the indentation of the tag is noted, and each line of code from the program is inserted at this point with the given indentation. The resulting string is then outputted to a file of the same name as the project with a .py extension. If no .pypc file was found, the lines of code from the Code Space are outputted once to the .py file. Run Code cannot be executed unless the project is loaded from/saved to a .pyp or .pypt file, but it notably does not require the project itself to be saved. This decision was made to encourage more exploration and run-edit cycles without the burden of needing to be confident enough in the changes of the project to save over the last version.

Once the Python script file is created, Python Pieces opens a new terminal and executes a "python" command to run the script with any arguments written in the Args box in the Menu Bar. One issue with working with an actual terminal was that it is unsafe to leave the terminal open after the script is executed, but if the terminal closes itself after the script terminates the user cannot see the output of the program. The solution was to have the terminal execute a very long "timeout" command (about 24 hours) right after the python command, and then close itself afterwards. This way the output of the program could be seen but no further commands could be executed from that terminal.

7. Future Work

Python Pieces functions well enough as a proof-of-concept currently, but there are a number of improvements that could be made to the project that would enhance the functionality and student-teacher friendliness of the product before distribution or publication.

- **Better aesthetics:** The option to choose and even save preferences on different fonts for the code and different background colors for the whole Workspace which might be more relaxing to the user's eye. The project would also benefit from more research and on optimal colors for both the text and the background of Code Blocks.
- **Visual cues for drop location:** Another nice improvement would be visual cues while dragging for where Code Blocks will be placed or moved to when released. A simple blinking cursor line, like a traditional text editor, that appears between existing Code Blocks to indicate potential insertion location would suffice to provide the user certainty on what will happen when they release the touch or click.
- **Integrated console:** Most IDEs, (such as Eclipse, Dr. Java, and PyCharm) have an integrated console. Kivy does not naturally support shell-like interactions, but an integrated console would be much more convenient, aesthetically pleasing, and safer to use than an external shell.
- **Better file system:** Currently, .pypt and .pypc are linked to their associated project files only by file name; if a student renames their project, then Python Pieces will not recognize the template files and not be able to Reset to Template or insert the project script into the true template linking. This seems particularly problematic for situations where students may want to make several versions of their project. In order to allow renaming without losing links, the file format could be updated to contain links to the template files that are independent of the project's file name. Another improvement would be the ability to to open .pyp and .pypt project files in Python Pieces with a double click from a typical file browser.
- **Safer and more convenient code integration:** The current method for inserting the project code into the .pypc template is simple to use, but has some major limitations. As noted above, it requires the external code that will be both spliced and run to be in a file of the same name as the project, which is potentially unsafe. It can also be cumbersome if the external code is already in another file or series of files. This might be improved by having a more explicit way for teachers to specify which scripts to insert the project code into and which script to run. Also, error messages refer to line numbers in the script with external and project code together, not the project code, which makes debugging more difficult for students. The goal is a more secure and flexible system for integrating student and external code that is still convenient.
- **Nesting Code Blocks:** Currently, Code Blocks sit next to each other in a Code Line. However, many common Python structures are cumbersome in this setting and would more naturally be expressed if nesting Code Blocks were possible. That is, a Code Block that is capable of containing some arbitrary number of Code Blocks within it at some point, like a "fill-in-the-blank." A few good examples for this are function call parentheses, list brackets, or dictionary braces, which syntactically require a matched pair. This would reduce syntax errors and visually promote understanding of the Python language. The primary difficulty with implementing this is how to handle text wrapping. Currently, if a Block cannot fit into the Code Line, it does not get split

but is rather pushed entirely to a second row within a Code Line in an attempt to gracefully wrap the Line; unfortunately, if the Block is still too long for the Line on its own row, it gets cut off visually. If a Block can grow arbitrarily long because of nested Blocks, the issue of how to gracefully handle wrapping is compounded.

- **Special Code Block interaction rules:** Another improvement could allow teachers to create special rules of usage for specific Blocks to help reduce syntax errors and encourage good style. For example, there could be special one-line blocks that take up a whole Code Line. Another possibility are adjacency rules that require or forbid other blocks being on their left or right, which could be applied to things like unary or binary operators. The Run Code button would perform a check on the blocks to see whether the rules are abided by before creating the .py file and running it.
- **On-screen Instructions:** One of the features of Codecademy is a side panel that contains instructions for the assignment. Since currently a teacher using Python Pieces would need to deliver instructions in some other form (such as oral instruction, text document, or physical handout), Having a panel that displays teacher-provided instruction specified for the project template would create a more self-contained learning experience.

8. Conclusion

Python Pieces ultimately helps teach a real programming language in an approachable visual environment designed to facilitate a teacher's own incremental lesson plan. Students writing programs with Code Blocks will find the programming experience to be less about fixing frustrating errors and more about exploring effective ways to solve problems with code. Teachers can customize the student's problem space for each lesson and take advantage Python Pieces' external code integration to make feedback more exciting for their students. Python Pieces represents progress in the area of computer science education tools for young learners, and with a few improvements Python Pieces can be ready for distribution and use in the classroom.

References

[1] kivy.org/#aboutus, accessed: 2015-04-30.

[2] www.kodugamelab.com, 2009-2015, accessed: 2015-04-30.

[3] www.coursera.org, 2015, accessed: 2015-04-30.

[4] A. Bradbury, "Pycon uk 2014: Teaching children to program python with the pyland game," www.youtube.com/watch?v=LTX4IHDNIcM, oct 2014, accessed: 2015-04-30.

[5] A. Bradbury *et al.*, github.com/pyland, 2014, accessed: 2015-04-30.

[6] www.codecademy.com, Codecademy, 2015, accessed: 2015-04-30.

[7] F. Georgatos, "How applicable is python as first computer language for teaching programming in a pre-university educational environment, from a teacher's point of view?" Master's thesis, Universiteit van Amsterdam, Amsterdam, jun 2002.

[8] P. J. Guo, "Why python is a great language for teaching beginners in introductory programming classes," pgbovine.net/python-teaching.htm, 2007, accessed: 2015-04-30.

[9] scratch.mit.edu/about/, Lifelong Kindergarten Group at the MIT Media Lab, accessed: 2015-04-30.

[10] appinventor.mit.edu, Massachusetts Institute of Technology, 2012-2015, accessed: 2015-04-30.

[11] www.python.org/community/sigs/current/edu-sig/, Python Software Foundation, 2001-2015, accessed: 2015-04-30.

[12] C. Wilson *et al.*, "Running on empty: The failure to teach k-12 computer science in the digital age," The Association for Computing Machinery and The Computer Science Teachers Association, 2010, from runningonempty.acm.org.

[13] J. M. Zelle, "Python as a first language," in *Midwest Computer Conference*, 1999.

Acknowledgements

- First and most understated thanks goes to my advisor Dr. Christopher Moretti for his unflagging positivity and flexibility.
- A grudging nod to Evan Miller for motivating me to work over spring break. You’re welcome for the burger.
- Eternal gratitude to Michelle-Ann Tan for keeping me company during the late nights of writing this paper.
- You, the reader, for your interest in both the worthy topic of CS education and my humble contribution to it.
- All thanks ultimately goes to Christ, who is my sole source of hope and worth on both my best days and my worst.
  Soli Deo Gloria.