Extending the Design of a Blocks-Based Python Environment to Support Complex Types

Matthew Poole
School of Computing
University of Portsmouth, UK
matthew.poole@port.ac.uk

***Abstract*—We are currently developing PyBlocks, a blocks-based environment which allows novice programmers to construct and execute Python programs. In the initial design of PyBlocks [1], Python's basic data types and lists are represented using colors, every expression block is colored according to its type, and each unfilled slot contains color indicating all valid argument types. In this paper we extend the design to include Python's most common built-in composite types (lists, tuples, dictionaries and sets) and to allow nesting of these where appropriate. Using example types from a pedagogical media computation library, we also show how further types may be supported. Together, these extensions provide almost any type novice Python programmers are likely to use.**

I. Introduction

Blocks-based languages such as Scratch [2], Snap! [3] and Blockly [4] offer several advantages over traditional text-based languages for novice programmers. However, many learners will at some point need to make the transition to traditional textual languages such as Java or Python. Other students' first taste of programming will be in such a textual language.

Whatever their prior programming experience, these novice programmers need to cope with the complexity of a mainstream language's syntax and semantics. One way to scaffold such learning is by providing a blocks-based environment for constructing programs directly in the text-based language of choice. In such an environment, the blocks themselves contain code in the target language and when connected together they form a program which can be read directly from the blocks. In [1], we gave initial design ideas for a blocks-based environment, PyBlocks, for Python 3. Python is targeted due to its comparative simplicity and its popularity as an introductory text-based language both in high school and in higher education. The current prototype version of PyBlocks implements this design. It has been built as a modification of Blockly and uses Skulpt, an in-browser implementation of Python [5], for program execution.

PyBlocks aims to achieve some of the benefits of blocks-based environments for learning Python, primarily the ability to browse the language and the ease of constructing programs free from syntax errors [6], [7]. PyBlocks also aims to reduce run-type errors, rare in block-languages but frustratingly common in languages such as Python which has a rich and dynamic type system. PyBlocks ensures that constructed programs will remain well-typed during execution by enforcing static typing. To help learners understand the use of types, an approach to type visualization based on color is used: each basic type is represented by a different color, every expression block is colored according to its type, and unfilled slots in blocks contain colors indicating all valid argument types.

The intended users of PyBlocks are high school or higher education students who are beginning to learn programming in Python, either with no previous programming experience, or are transitioning from having used a blocks-based programming environment such as Scratch. PyBlocks could be used as a first environment for editing Python programs, or as a fallback for students who are struggling with the structure of the Python language when faced with a text editor. PyBlocks doesn't aim to support the whole Python 3 language; rather, our focus is on providing those features commonly taught in introductory courses in procedural programming.

Because of the complexity of Python's type system, the initial design in [1] was restricted to supporting Python's four "basic" types (integer, float, string and Boolean) and lists of these. Although many programs can be written just using these types, many more can't; the provided types may appear particularly limiting for students in higher education. In order to greatly expand the possibilities for what novice programmers can learn and create, we need to allow for more complex types. Firstly, in addition to lists, we need to include Python's other commonly-used composite data types—tuples, dictionaries and sets—and to allow for nested structures. Secondly, we need to provide support for user-defined types from, for example, popular libraries for graphics [8] or media computation [9].

This paper is structured as follows. In the next section we discuss related work. In Section III we review the principles that have guided the design of PyBlocks and how it works with basic types, as detailed in [1]. Section IV describes how composite types are represented and in Section V we show how non built-in library types can be added. Section VI concludes the paper and discusses future work.

II. Related Work

Other work on blocks-based editing of textual languages includes Tiled Grace [10] for the pedagogical language Grace, and Pencil Code [11], [12] which supports JavaScript and CoffeeScript. In both systems the user can edit a program directly using blocks and also switch to a well-formatted editable text view. Expression types are generally not explicitly indicated and restrictions on what constitutes a legal block argument often only becomes apparent on attempting a drag-and-drop operation.

Representing types as colors is not common in blocks-based environments. Probably the use of color most related

978-1-5386-2480-7/17/$31.00 ©2017 IEEE

to that of the current paper is in the prototype blocks editor for Bootstrap [13], [14]. This functional language's five types are each represented by a color. A neutral color (gray) is used for polymorphic blocks such as the conditional expression, and polymorphic blocks change color once their type has been determined during program construction. Being focused towards middle school use, the language is relatively simple and no complex types are supported.

There have been efforts to support complex types in block languages, and these typically concern the design of connector shapes. TypeBlocks [15] includes three basic type connectors which can be combined in any way and to any depth using type constructors for lists, pairs and functions. Each constructed type has its own unique connector shape. Polymorphic Blocks [16] takes a similar approach but also supports parametric polymorphism through the use of "polymorphic ports". A block's polymorphic ports are colored rather than shaped, and when a port is connected to a shaped connector of another block, all ports of the same color become (uncolored) connectors of that shape. Both TypeBlocks and Polymorphic Blocks provide a complete visual representation of arbitrarily complex types using connector shapes. As the types get more deeply nested however, the connectors can become very intricate and blocks necessarily need to grow in order to allow their types to be discerned.

III. Overview of PyBlocks

Block-based environments aim to reduce "syntax overload", and to guide the user in constructing syntactically legal programs. This is also a primary goal of PyBlocks. Learners of dynamically typed text-based languages also know that a major source of frustration is the frequent occurrence of run-time errors—students are constantly faced with "TypeError" messages (such as when a program attempts to add a float and a string). Block languages such as Scratch tend to avoid run-time errors, but do so by means of simpler and more forgiving type systems than Python's.

PyBlocks minimizes type-related run-type errors by enforcing static typing during the construction of programs. This is achieved by requiring that (i) variables' types are fixed ("declared") when they are created within the palette, and (ii) valid argument types of all operator and function blocks are enforced. At the same time, the types of all blocks and slots are made explicit to the user to help them learn how types work within their programs. This latter point tends to be de-emphasized in typical block languages.

The Python code in a PyBlocks-constructed program conforms to Python's rules and conventions regarding formatting and whitespace, demonstrating good code layout to learners and also helping with a more seamless transition to textual code editing [17]. Use of shaped connectors would break whitespace conventions and leads to awkward vertical jumps within lines of code. Furthermore, the richness of Python's type system would would require many, probably unwieldy, connector shapes. PyBlocks avoids shaped connectors and instead uses colors to denote basic data types: every expression block is colored according to its type, and each unfilled slot indicates, using color, all valid argument types.

A. Type colors and literals

Colors for the four basic (built-in, non-composite) types are illustrated in Fig. 1 using blocks for literal values. Strings are considered basic since they are not built from another type (there is no character type in Python). The absence of space around the values, and the lack of rounded corners helps with adherence to conventions concerning whitespace.

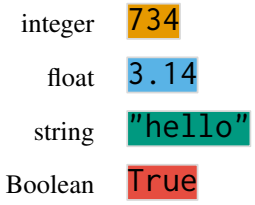

Fig. 1. Basic type colors and literal values

These four colors have changed from those used in [1]; the new colors are adapted from a palette of colors designed to be perceived as distinct by users who are color blind and those who are fully sighted [18]. (Note that any literal value within a block can be edited by the user but only to a value of the same type.)

B. Function and operator blocks

Blocks for functions and operators are colored according to their result type and, to make clear the valid argument types for these blocks, all unfilled slots are marked with "type indicators" showing all acceptable argument types.

Example blocks are shown in Fig. 2. The integer division operator "`//`" is colored orange to denote that it gives an integer result. The two slots' type indicators are also colored orange since the arguments should also be integers; any attempt at dropping an invalid block into either slot will be rejected. The `float` block is sky-blue (it returns a float). Its slot indicator is striped orange/green, meaning that it will accept any integer or string block, or a multicolored block that includes either or both of these colors—a block only needs to *share one color* with a slot indicator to *match* it. The addition operator block is striped sky-blue and orange, as are both its slots' indicators—it returns either a float or an integer and this type depends on the types of its arguments, which can also be floats or integers.

Fig. 2. Blocks for the integer division operator, the `float` function and the addition operator

Fig. 3 illustrates some effects of dropping blocks. Dropping the float block (2.3) into a slot in the addition block causes the addition block to become solely sky-blue: with a float operand the result of an addition can only be a float. Dropping an empty addition block into a slot of the integer division block has three effects: (i) the addition block becomes orange since it must supply an integer to the division block, (ii) both of its indicators become orange to ensure its arguments can only be integers, and (iii) a pair of parentheses is added to ensure correct order of evaluation (parentheses are added

automatically and only when they are required). Note that if a nested block is subsequently removed from a slot, then all affected blocks are recolored appropriately.

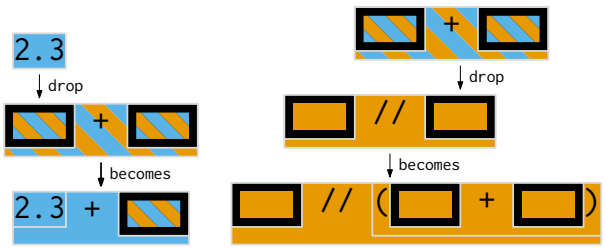

Fig. 3. Dropping of blocks into slots leading to recoloring of blocks and slot indicators, and the appearance of parentheses

C. Statement blocks

Statement blocks do not have result types and are therefore colored a neutral gray. Statement blocks in the current version of PyBlocks include assignments, `if`-statements, `for` and `while`-loops, and blocks that call functions and methods, such as `print`, which do not return values. Statement blocks are chained together and nested using a simple "notch" connector. Fig. 4 shows an incomplete program fragment containing assignment, `print` and `if`-statement blocks.

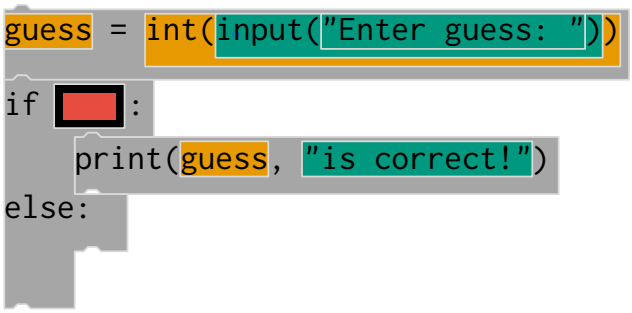

Fig. 4. An incomplete fragment of Python code in PyBlocks

We note that some blocks can be modified (by clicking on the block to give a context menu) in order to change the number of connecting or argument blocks; for example, `elif` and `else` clauses can be added to an `if` block, and argument slots can be added to a `print` block.

IV. Composite types

Python includes a rich set of several built-in composite types, primarily:

i *lists* – ordered, mutable collections;
ii *tuples* – ordered, immutable collections;
iii *dictionaries* – unordered, mutable key-value mappings;
iv *sets* – unordered, mutable collections.

All these types are widely used by experienced programmers. They also feature in introductory programming textbooks and courses, and occasionally they are used in combination (i.e. nested). In [8], for example, all types except sets feature, and tuples are nested within lists. The same is true of [19] where multidimensional lists are also used.

In deciding which of the types to support within PyBlocks, and to what extent to support nesting, we need to reach a trade-off between how likely a given type or combination of types is to be used by a novice programmer, and how these types can be visually represented within blocks.

A core aim of PyBlocks is to provide a complete visual representation of the type of every expression block and the allowable types for empty slots. These representations need to be both visible and comprehensible. Blocks cannot stretch horizontally to accommodate more complex visual representations, since this would break adherence to whitespace conventions. (In rare cases a variable of some complex type could be forced to have a name of minimal length in order to accommodate the visual representation of its type.) Blocks can, however, grow downwards. In fact, all information concerning the type of a function or operator block will typically need to appear in the bottom portion of the block, below its argument slots.

Python's dictionary keys and set elements must be of *hashable* type, and the only built-in types that are hashable are those that are immutable (note that tuples are only hashable if all their elements are). This means that many combinations of composite types (such as sets of lists) are illegal. Other nestings are legal but of little practical value (e.g dictionaries as values of dictionaries). Legal nesting of composite types within tuples, dictionaries and sets is generally rare and we disallow them (although see a special case in Section IV-H). Nesting of lists is, however, quite common. In fact, supporting nesting (of any composite type) only within lists covers almost any conceivable use by a novice programmer; lists of lists and lists of tuples are particularly common. It is relatively easy to allow an unlimited depth of list-list nesting (very deep nesting just leads to very tall blocks) and to allow any another composite type to appear at the deepest level (e.g. list of list of tuples).

A. Rainbow coloring

The representations of the various composite types use different white markings combined with colors representing the element type(s) they contain. Central to the design of expression blocks for these types are the concepts of "any basic type" and "any type". Our use of colors for basic types and color striping for alternative types (as in Fig. 2) suggests the use of rainbow stripes to stand for any basic type. A rainbow with white stripes pattern is used to denote any type, including basic types, composite types and indeed any constructible nested composite type. These two patterns are illustrated in Fig. 5.

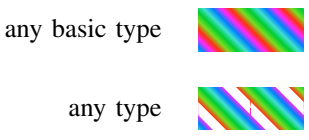

Fig. 5. Rainbow and rainbow-and-white type patterns

Rainbow-and-white coloring is illustrated by the assignment block in Fig. 6; both slot indicators are rainbow-and-white meaning that any argument type is permitted. The left-hand slot is also marked "`var`" to signify that only variable

blocks may be dropped into it. On dropping a string variable block into this slot, the indicator in the right-hand slot is recolored accordingly (to enforce static typing).

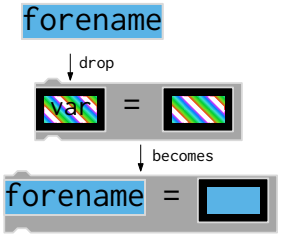

Fig. 6. Assignment with rainbow-and-white patterns and "var" indicator

We note that the two rainbow patterns also match themselves and one another. So, the rainbow-and-white pattern matches with everything; the rainbow pattern matches the rainbow-and-white pattern and everything else that doesn't include white.

In the following sections we see how lists, tuples, dictionaries and sets are represented visually and how blocks of these types behave during editing. None of these types match with any of the others (e.g. a list cannot match with a tuple) and none match with any of the basic types. Within each category of composite type, matching depends on the type of elements they contain.

B. Lists

PyBlocks ensures that all lists are homogeneous to encourage good programming style and to allow for the benefits of static typing [1]. The original design for lists (featuring two vertical bars) has been replaced by saw-tooth markings at the bottom of the block; see Fig. 7. The revised design allows us to more easily represent nested lists and the other composite types. Two list types match only if their content types match (e.g. if they contain a common color).

Fig. 7. List variable blocks showing a list of integers in the original design and a list of floats and list of strings using the revised design

Fig. 8 illustrates two list operator blocks. The list constructor block on the left has a rainbow-and-white slot indicator accepting any type; the block itself gives a list (saw-tooth at the bottom) of any type (rainbow-and-white). Note the use of the gray line on the edge of the saw-tooth that provides a clear boundary between it and the rainbow-and-white pattern. Dropping an integer block into the slot causes recoloring of the block. The first slot in the list indexing block on the right accepts a list containing values of any type and the block itself can return any type. Dropping a list-of-string block into the slot recolors the block to green (indexing a list of strings results in a string).

C. Nesting of lists within lists

The saw-tooth pattern adopted for lists allows for a straightforward visual representation of nested lists. Fig. 9 shows a two-element list of floats being dropped into a slot of an

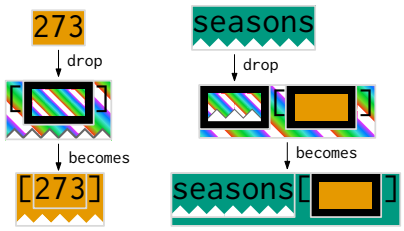

Fig. 8. Behavior of the list construction and list indexing blocks

empty list construction block. The resulting block includes a double saw-tooth pattern representing the type list-of-list-of-floats, with gray used on the lower saw-tooth (outer list) to make the representation clear. This stacking of saw-tooth patterns allows any depth of nesting (with an increase of block and indicator height) and, importantly, the depth of nesting (at least up to a few levels) can be readily identified by the user.

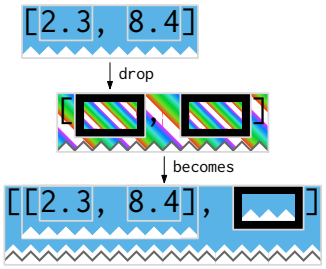

Fig. 9. A list of floats is dropped into a slot of a list literal block resulting in a nested list

D. Tuples

Tuples in Python are heterogeneous immutable collections. Their use is typically limited to combine data to form "records", such as `(25, True)`, of pairs or triples. Elements of tuples are mainly accessed via unpacking (see below). To allow for static typing, we prohibit iteration through the elements of tuples elements (tuple iteration is rarely used in Python). We also limit tuples' elements to be of basic type, preventing overly complicated structures and representations.

Tuple types are represented within blocks with the element type colors appearing in order from left-to-right, separated by white columns. Two tuple types match only if they have the same number of elements and, for each element, the two element colorings match. A tuple construction block for a pair is illustrated in Fig. 10. The rainbow patterns in the indicators restrict the arguments to basic types. (Note that parentheses around tuples are not always needed but are very commonly used.) Unlike for (homogeneous) list construction, dropping a block (an integer in the figure) into one of this block's slots does not cause recoloring of the other slot's indicator. We see that the white column is positioned to align with the comma so that it is fully visible and shows clearly the relationship between the types of the tuple and those of its components.

Tuples can be assigned to variables in Python in two different ways. Firstly, they can be assigned to variables of the same tuple type. Alternatively, their elements can be "unpacked" and assigned to separate variables using a multiple assignment statement; see Fig. 11.

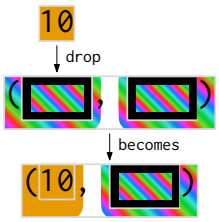

Fig. 10. Using the tuple construction (comma) operator block

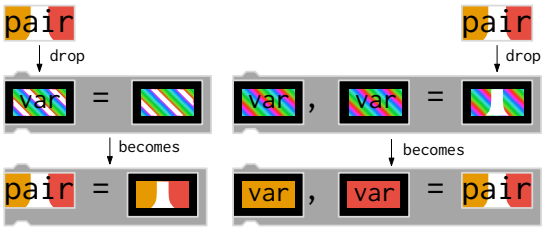

Fig. 11. Dropping a tuple variable block into the left-hand side of an assignment block and the right-hand side of a multiple assignment block

E. Dictionaries

Python dictionaries are mappings from keys to values, where the keys must be of hashable type; an example dictionary literal is `{"jam": 1.65, "fish": 2.89}`. We will restrict dictionaries to be homogeneous (keys all of one type, values all of one type) where both keys and values are basic types.

We represent dictionary blocks with two areas of color, the key type represented on the left, and the value type on the right, separated by a vertical saw-tooth bar "pointing" from left-to-right. Two dictionary types match if the colorings for keys and values both match.

Fig. 12 illustrates the construction of a dictionary literal of a single (string–float) pair; this is then dropped into the right-hand-side of an assignment block.

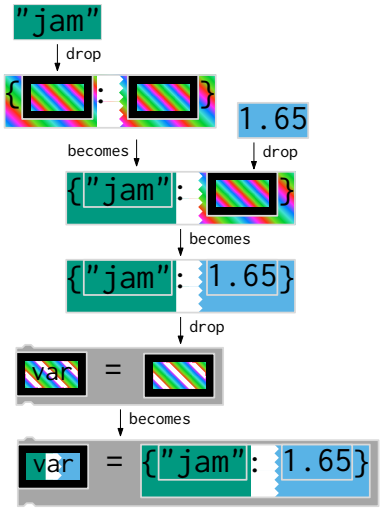

Fig. 12. Construction of a single-element dictionary literal which is then dropped into the right-hand side of an assignment block

F. Sets

Python sets are unordered collections of hashable data values; `{3,7,6}` is an example set literal. Our design restricts set contents to be of basic type and homogeneous. We use semi-circular markings at the bottom of the block to denote sets. Two set types match only if their element types match. Fig. 13 shows the construction of a two-element set of strings, which is then dropped into the right-hand side of a set "`in`" membership testing block.

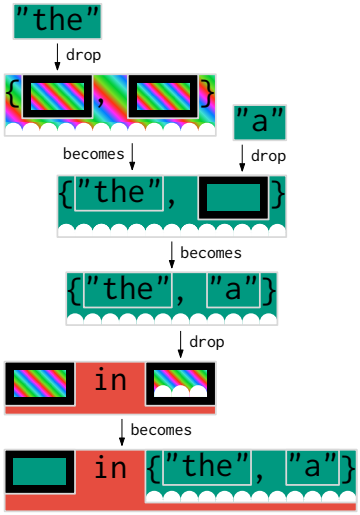

Fig. 13. Construction of a set-of-strings literal which is then dropped into the right-hand side of a membership test block

G. Lists with composite elements

As discussed above, tuples, dictionaries and set blocks cannot generally be nested within one another. However, our design does allow lists (nested to any depth) containing these structures. Fig. 14 illustrates the construction of single-value lists containing a tuple, a dictionary and a set, respectively. Again note the use of gray on the top edge of the outer type representation (the list saw-tooth).

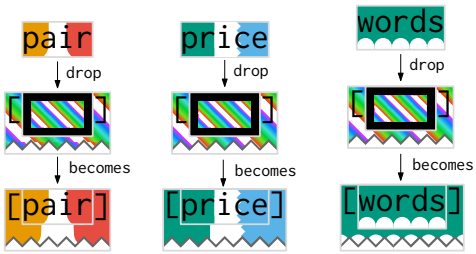

Fig. 14. Construction of singleton lists of tuples, sets and dictionaries

H. Functions returning multiple values

It is quite common for a user-defined function in Python to return multiple values. These values are in fact returned within a tuple and then almost always unpacked by the calling code into separate variables. Consider, for example, a function `countVowels` which takes a string as a parameter, and returns two values: a dictionary giving the frequency of each vowel in the string, and a list of all non-vowels found in the

string. The resulting tuple type, (dictionary, list), would not normally be permitted by our design where elements of tuples need to be of basic type. To overcome this shortcoming, we will allow tuples with elements of any type to appear within `return` statement blocks.

If PyBlocks then provided an expression block for invoking the function `countVowels`, this block would also be of type (dictionary, list). It could then potentially be dropped into any rainbow-and-white colored slot of another block, possibly leading to an even more complicated type. To prevent this from happening, we propose that PyBlocks does not create blocks for invoking functions such as `countVowels` that return tuples with composite elements. Instead, PyBlocks will only provide code that invokes such functions "baked-in" to the right-hand side of special tuple unpacking assignment block. Fig. 15 shows the `return` block and function call unpacking block for `countVowels`.

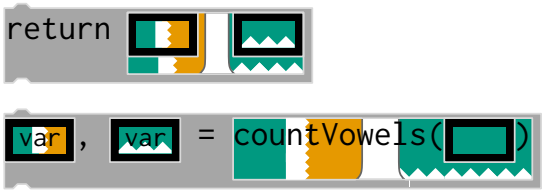

Fig. 15. A return block for a function returning a dictionary and a list, and a block for calling this function and unpacking the results.

I. *For loops and the range type*

Python's `for` loop is used to iterate over any *iterable* type, which includes strings, ranges and lists. Rather than designing a single indicator to encompass all of these types, we instead show three alternatives type indicators in the slot; a block needs to match any one of these to be dropped into the slot; see Figure 16. The range type in Python 3, which is represented as a composite type containing integers, exists mainly for use in `for` loops. Dropping the single-argument `range` block into the loop slot causes the recoloring of the loop variable slot indicator.

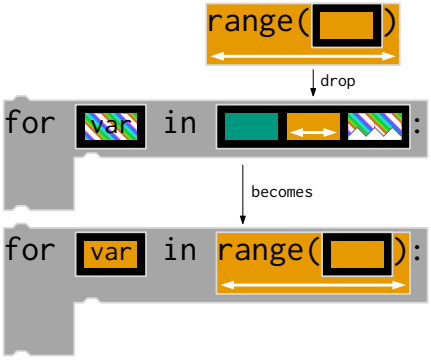

Fig. 16. A `range` block and a `for` loop block with three indicators denoting alternative valid argument block colorings

V. Providing other types

Although the addition of composite types clearly expands the sorts of programs that can be written, learners nowadays should expect to be able to produce more interesting applications than are permitted by Python's built-in types alone. Educators who use Python have recognized this and have developed learner-friendly libraries containing multiple types for writing non text-based programs; examples include Zelle's graphics module [8] and Guzdial and Ericson's Media Computation [9].

There are two challenges in integrating these libraries into PyBlocks. One concerns program execution: many libraries have been developed for desktop use, rather than web use. However, there are efforts to create web-based versions of these particular libraries to work with Skulpt; a version of media computation has been developed for Pythy [20], and we have developed a prototype version of Zelle's graphics module.

Perhaps the more obvious problem that concerns us here is the number of types provided: there are 11 types defined in the graphics module and 8 in the media computation library. Few users, particularly those who are color blind, would be able to differentiate between so many colors in addition to those used for the basic types.

To address this, rather than using a separate color for every type, we instead use a single new color to cover all the types from a particular library. We then introduce a symbol to identify each of the library's types, and embed this symbol within blocks and type indicators. A similar use of symbols to represent types within blocks and indicators is found in the Waterbear block editor [21].

Fig. 17 illustrates the use of blocks involving the media computation types `Picture`, `Pixel` and `Color`. Here, `makePicture` takes a filepath (a string) and returns a `Picture` value; `getPixels` returns a list of `Pixel` values from a `Picture`; `getBlue` returns the blue element of the color of a `Pixel` as an integer; and `setColor` sets a `Pixel` to a given `Color` value.

Fig. 17. Example media computation blocks

Fig. 18 shows a short program using the media computation blocks, illustrating how the types of all expressions in a program are complete, clear and fully determined (no multi-colored or rainbow patterns remain).

Whilst it would be relatively straightforward for non built-in types such as these to be considered as basic types, we currently see little benefit of this—their use within dictionaries, tuples and sets seems much less likely than in lists. Also, the use of white for the symbols contrasts well with the block color. Given this coloring scheme, considering these types as non-basic maintains the simplicity of the rule for the rainbow pattern: i.e. that it matches the rainbow-and-white pattern and everything that doesn't include white. The rainbow-and-white pattern still matches everything.

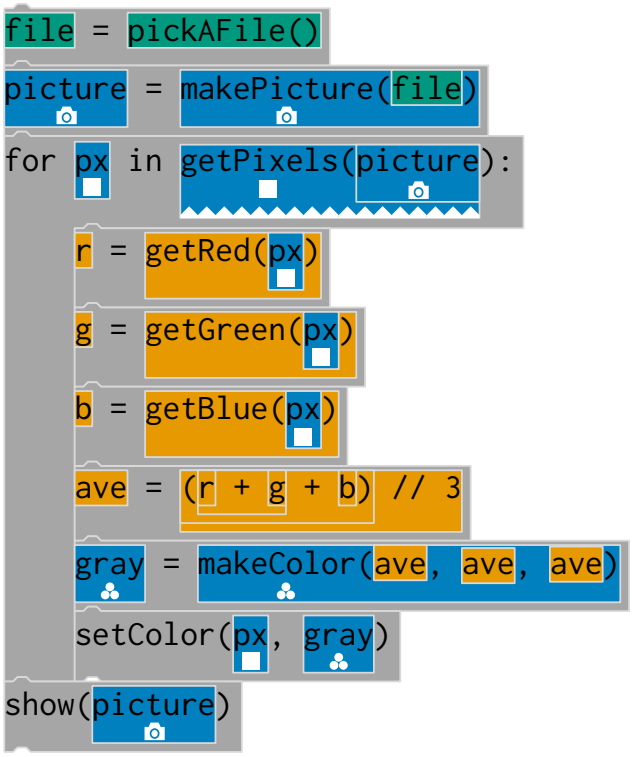

Fig. 18. A complete media computation program for displaying a picture from a user-selected file in grayscale

VI. Discussion and Conclusion

We have extended the initial design of PyBlocks from [1] to provide support for Python's built-in composite data types and types from user-defined libraries. Together, these additions should cover the vast majority of types used in typical learning environments.

Although arguably a good introductory language, Python is designed primarily for professional programmers and this is reflected in a rich type system which is more complex than those of most block languages. The decisions and trade-offs made here attempt to increase substantially the number of useful types provided whilst preserving a complete and comprehensible type visualization that is usable to the learner.

Representing complex types visually within blocks is difficult. Attempts using block connectors, in [15] and [16] for example, can lead to connector shapes that are visually complicated and which require blocks to grow significantly in order to accommodate them. The representations discussed in this paper are certainly not all simple. However, by placing reasonable limits on the types that can be represented, and by using a combination of a few colors (for basic types), white markings (for composite types) and symbols (for library types), we believe the visual representations compare favorably with related efforts.

Future work will include implementation of these design ideas within PyBlocks together with the integration of pedagogical modules and libraries including those discussed above. We also plan to test the ideas presented via software trials with users from the target groups.

References

[1] M. Poole, "Design of a blocks-based environment for introductory programming in Python," in *Blocks and Beyond Workshop*. IEEE, 2015, pp. 31–34.

[2] "Scratch," scratch.mit.edu, accessed 17 July 2017.

[3] "Snap!" snap.berkeley.edu, accessed 17 July 2017.

[4] "Blockly," developers.google.com/blockly/, accessed 17 July 2017.

[5] "Skulpt," www.skulpt.org, accessed 17 July 2017.

[6] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: blocks and beyond," *Communications of the ACM*, vol. 60, no. 6, pp. 72–80, 2017.

[7] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 199–208.

[8] J. Zelle, *Python Programming: An Introduction to Computer Science*, 3rd ed. Franklin, Beedle & Associates Inc., 2016.

[9] M. J. Guzdial and B. Ericson, *Introduction to Computing and Programming in Python, A Multimedia Approach*, 4th ed. Prentice Hall Press, 2016.

[10] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE, 2014, pp. 1–10.

[11] D. Bau, D. A. Bau, M. Dawson, and C. Pickens, "Pencil code: block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 445–448.

[12] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, 2015.

[13] "Bootstrap block editor," bootstrap-block-editor.appspot.com, accessed 28 August 2017.

[14] E. Schanzer, S. Krishnamurthi, and K. Fisler, "Blocks versus text: Ongoing lessons from Bootstrap," in *Blocks and Beyond Workshop*. IEEE, 2015, pp. 125–126.

[15] M. Vasek, "Representing expressive types in blocks programming languages," Wellesley College, Honors thesis, 2012.

[16] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic blocks: Formalism-inspired UI for structured connectors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 3063–3072.

[17] M. Kölling, N. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM, 2015, pp. 29–38.

[18] B. Wong, "Points of view: Color blindness," *Nature Methods*, vol. 8, p. 441, 2011.

[19] B. N. Miller and D. L. Ranum, *Python programming in context*, 2nd ed. Jones & Bartlett Publishers, 2014.

[20] 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*. ACM, 2014, pp. 641–646.

[21] "Waterbear," waterbearlang.com, accessed 28 August 2017.