Design of a Blocks-Based Environment for Introductory Programming in Python

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

***Abstract*—This paper details the design of a visual blocks-based tool for editing Python programs. Its purpose is to close the gap between programming using a simplified blocks-based language and textual programming in a mainstream language. As well as helping to guarantee the syntactic validity of programs, the tool aims to reduce the occurrence of run-time errors, a source of learner frustration with dynamic languages, by ensuring that constructed programs will remain well-typed during execution. The design promotes understanding of how data types are used in the language by representing them using colors: each expression block is colored according to its type, and each unfilled hole contains colors which indicate valid argument types. Connected blocks preserve conventional use of whitespace, demonstrating good practice for novice programmers.**

I. Introduction

Visual programming systems such as Alice [1], Blockly [2] and Scratch [3] are attractive environments in which to learn to program. They each provide a restricted programming language and an accessible editing environment of jigsaw piece program elements manipulated through drag-and-drop interactions.

Many learners will need to move onto traditional textual languages such as Java, Python or JavaScript. 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 combination of the complexity of a mainstream language's syntax and semantics and the absence of any guidance provided by blocks-based editing.

Python is an increasingly popular language in introductory programming courses due largely to its comparative simplicity. This paper describes a design for a blocks-based environment for Python 3, with the aims of providing the advantages of visual editing to support the learning of programming in a mainstream textual language.

This paper is structured as follows. In the next section we detail the aims and principles which have guided the design. Section III describes the design of the blocks themselves, and Section IV demonstrates how programs are constructed. Section V discusses future work and concludes the paper.

II. Design Principles

The target users of the proposed system 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. The intention is for the tool to be used as a first environment for editing Python programs, before moving on to a text-only IDE such as IDLE. It could also be used as a fallback for students who are struggling with the structure of the Python language when faced with a text editor. The design doesn't aim to support the whole Python 3 language; instead it focusses on those features commonly taught in the first phase of an introductory course. The current paper describes support for constructing programs featuring expressions, assignments, input-output and control structures. A future paper will consider function definitions, module imports and user-defined classes, in addition to giving implementation details.

By their very nature, an aim shared by block environments is to reduce "syntax overload", and to guide the user in constructing syntactically legal programs. This is also a primary aim of the proposed software. Learners of mainstream dynamically typed languages also know that a major source of frustration is the frequent occurrence of run-time errors (one disadvantage of their use compared with statically-typed languages such as Java). Students are constantly faced with "TypeError" messages (such as when a program attempts to add a float and a string or to index an integer). Block languages such as Scratch tend to avoid run-time errors, but do so by means of simpler, more forgiving and less dynamic type systems than Python's.

The proposed software aims to minimize such type-related run-type errors, mainly 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 block palette, and (ii) valid argument types of all operator and function blocks are enforced. Furthermore, the types of all operations, arguments and expressions should be made explicit to the user to help develop their understanding of how types work within their programs. This latter point tends to be de-emphasized in typical block languages.

Programs in textual languages should not only be correct, but should also be readable; readability necessitates adherence to accepted conventions regarding whitespace. The use of indentation in Python helps to enforce readability, but correct whitespace between neighboring lexical elements (e.g. around operators and after commas) is also important. A block editor which honors correct whitespace would not only demonstrate good code layout to learners, but would also help provide a more seamless transition to textual code editing. The use of connector shapes in languages such as Blockly and Scratch

takes up space on the left/right sides of blocks and thus prevents conventional use of whitespace.

Some recent related work on editing textual languages using blocks includes Tiled Grace [4] and Pencil Code [5], [6]. Tiled Grace supports the pedagogical language Grace, whilst Pencil Code currently works for JavaScript and CoffeeScript. In both systems the user can edit a program using blocks and then switch to a well-formatted text view, and in Pencil Code whitespace is preserved in the block view. Colors are used in both systems, but somewhat arbitrarily and not for types. 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, if at all.

III. Block Design

The presented design supports those core Python types which might be used in a typical introductory programming course. These types include integers, floats, strings and Booleans, and lists of these basic types. Ranges are supported for their use in `for` loops. Tuples and dictionaries are not currently supported in this design. It is envisaged that all the most commonly used built-in operators, functions and methods would be available for the user in a block palette with categories for statements, variables and for each supported type. Assignments, conditionals and loop statements, as well as comment lines would be present in the statement palette. This initial design does not describe module imports or function definitions, but clearly inclusion of these features would be desirable in a subsequent version.

Each type is assigned a unique color, and blocks are colored according to type. Statement blocks are considered uncolored, and are chained together vertically using a single block connector shape.

A. Type colors and literals

Colors for the four basic types are illustrated in Fig. 1 using blocks for literal values. The numeric and string blocks are editable to allow the user change the literal value within the same type. The absence of space around the values, and the lack of rounded corners, is deliberate due to the desire for programs to adhere to conventions concerning whitespace.

integer (orange)	`217`
float (red)	`3.14`
string (green)	`"hello"`
Boolean (blue)	`True`

Fig. 1. Basic type colors and literal values

B. List types

Our design also allows for homogeneous lists of the basic types. We use blocks with three colored stripes to represent list types, hinting at the notion of multiple values. Example list variable blocks are illustrated in Fig. 2. The choice to restrict the types of lists simplifies the color representation required, aids static typing, and also helps to enforce good programming practice.

Fig. 2. Two list variable blocks: a list of strings and a list of floats

C. Function and operator blocks

All blocks except those for literal values and variables will include holes for argument blocks. Blocks for operators and functions which give result values are colored according to the result type. Furthermore, to indicate clearly the valid argument types for these blocks, all unfilled holes are marked with small "type indicators" showing all acceptable argument types.

Example function and operator blocks are shown in Fig. 3. The `round` block is colored orange since it gives an integer result. The argument hole includes a single red type indicator, denoting that the argument should be a float. Any attempt at dropping a non-red block into this hole will be rejected. The `float` block is colored red (calling `float` results in an float value); the green and orange type indicators state that the argument should be either a string or an integer. Finally, we can see that the division operator '/' takes two numeric values (integers or floats) and produces a float.

Notice that operator blocks cannot extend horizontally past their arguments if we want programs to adhere to whitespace conventions. We therefore extend blocks downwards in order to make clear the type, extent and structure of expressions.

Fig. 3. Blocks for the functions `round` and `float`, and the float division operator '/'

The result type of many built-in functions and operators depends upon the argument types. Examples include functions such as `abs` (absolute value) and overloaded operators such '+' and '*'. Whilst `abs` operates only on numeric data, the symbols '+' and '*' are also used for string and list operations. We choose to have three distinct pairs of blocks (i.e. numerical, string and list) for these operators since they differ in nature for the different types. Fig. 4 shows the blocks for `abs` and the overloaded numerical operator '+'. With as yet unfilled holes, the result types of these blocks are not fully determined—each could return either an integer or a float—and they are therefore colored both orange and red, using a chequered pattern.

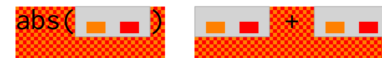

Fig. 4. Blocks for the function `abs` and the overloaded operator '+'

As we will see in Section IV, the colors of blocks and the visibility of type indicators is dynamic: as argument blocks are added, a multicolored block may become monocolored, and some type indicators may disappear from argument holes.

Some operations (e.g. the equality operator '==') accept arguments of several types. Rather than include a large number of type indicators in such a case, we use a single color, gray, to represent any of the four basic types, and striped gray to stand for any list type. Fig. 5 shows the block for '==', and also

blocks for list construction and list indexing, which together serve to illustrate the use of gray.

We see that both arguments to '==' can be of any basic or list type, and the result type is Boolean. List construction is considered an operator; an argument of a basic type will result in a list of that type. (The small triangle indicates a menu allowing the addition of extra arguments.) Indexing a list with an integer gives a value of the type of data stored in the list.

Fig. 5. Blocks for the equality operator '==', for list construction and for list indexing

The result types of most of Python's commonly used functions and operators, including all those discussed above, are either fixed or determined from the types of their arguments. In some cases however, we need to know the *values* of arguments. The most obvious example is the `eval` function which evaluates the contents of a string argument (`eval("12")` is the integer 12 and `eval("1 + 2.1")` is the float 3.1). In introductory programming, `eval` is typically used to convert keyboard (string) inputs into numerical values. However, the use of the functions `int` and `float` is preferred since (i) their use encourages students to think more about their choice of types, and (ii) they are considered simpler and safer. We choose therefore not to include `eval`.

The power function `pow` (and operator '`**`'), which when empty is colored like the '+' operator in Fig. 4, cannot reasonably be omitted. The `pow` function returns a float if either of its arguments are floats. If both arguments are integers then the return type depends on the value of the second argument: a non-negative value results in an integer, and a negative value gives a float. We take a straightforward approach to typing `pow` blocks if the arguments are both integer blocks: the `pow` block is typed (colored) as an integer, and if the second argument is not an integer literal (i.e. not guaranteed to be non-negative) then the user is alerted that the type will be incorrect at run-time whenever the value of the second argument is negative. (This is unlikely to cause a problem in practice: the second argument will most often be a literal value such as 2.)

D. Statements

The proposed design includes blocks for assignment, conditional and loop statements. Blocks for functions and methods which do not return values (including `print` and `append`) are also considered as statements. Statement blocks do not have types, are all colored neutral gray-green, and are chained together vertically using a simple 'notch' connector.

When the user introduces a variable, which involves choosing a name and a type, an assignment block and appropriately colored variable block are added to the block palette; see Fig. 6. The new variable might be considered to be 'declared' with the given type (albeit externally from the program text), and the assignment block will only allow it to be assigned values of this type.

The `if` and `while` statement blocks take Boolean-valued conditions, as do blocks for the Boolean operators such as `or`.

Fig. 6. Assignment and variable blocks for an integer variable

This prevents the use of data of other types being interpreted as Boolean values which often leads to confusion for novice Python programmers (for example, the expression `x == 10 or 20` is legal Python and always evaluates to True since 20 is interpreted as True). These blocks are illustrated in Fig. 7. The block for the `if` statement includes a triangle menu indicator for adding `elif` and `else` clauses.

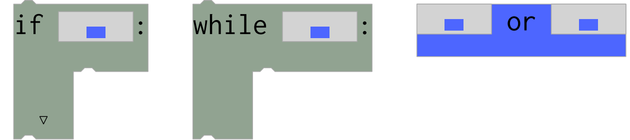

Fig. 7. Blocks for the `if` statement, `while` loop and Boolean `or` operator

The range data type and three `range` function blocks (for ranges with stop-values, and optional start- and step-values) are provided for use in `for` loops. The `for` loop block includes a selector to choose or create a loop variable of an appropriate type, and a hole with type indicators allowing a list, string or range argument block over which to iterate. Fig. 8 shows the `for` loop and the single-argument `range` block. Note the use of a new color, magenta, for the range type.

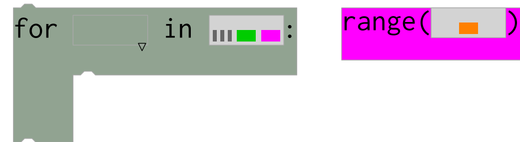

Fig. 8. Blocks for the `for` loop and `range` function

IV. PROGRAM CONSTRUCTION

Programs are constructed by dragging blocks from the block palette and dropping them into place within a workspace, as in Blockly. As a block is dragged, appropriately colored type indicators in close-by empty holes are highlighted to the user, and any attempted illegal drops are rejected.

Fig. 9 illustrates the interactions involved in the construction of a list indexing expression. Dropping the `rainfall` variable block (a list of floats) into the gray list indexing block causes the latter block to be recolored red (float), since indexing a list of floats will result in a float. Dropping the `day` integer variable block into the first argument hole of the '+' block causes no recoloring (the type of '+' will now depend on the second argument). The '+' block can be dropped into the list index hole (which has an integer type indicator), since the '+' block's colors include orange and so it can potentially be typed as an integer. When the drop is made, the '+' block is recolored to just orange, and the remaining red type indicator disappears to ensure that the second argument to '+' cannot be a float. Dropping the integer literal 1 block into this argument hole completes the expression. Note that, as subexpression blocks are added, the lower portions of operator and function

blocks are narrowed vertically. This ensures that the outermost expression block remains the same height and therefore that lines of code within a program are equally spaced. Blocks' heights would only need to increase if they were deeply nested.

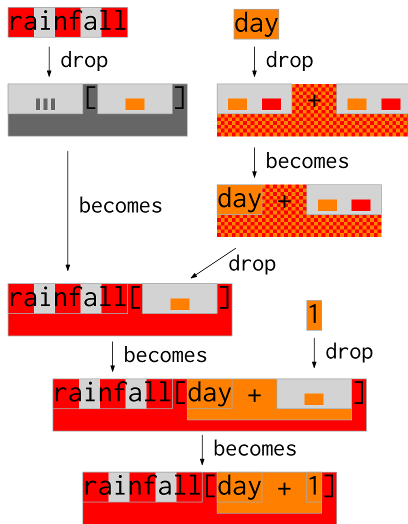

Fig. 9. Construction of an expression by drag-and-drop

One property of block languages is that parentheses are not needed to control the order of subexpression evaluation: the nesting of blocks suffices. For our purposes however, parentheses are required. Therefore, when an operator block is dropped into (i) an argument hole of a higher-precedence operator block, or (ii) the right-hand hole of an equal-precedence operator block, parentheses are automatically added around the dropped block; see Fig. 10.

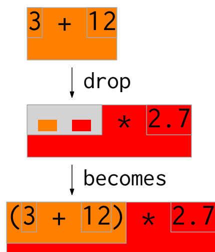

Fig. 10. Automatic insertion of parentheses

In the current design, a program is considered complete and valid by the editor if (i) it comprises a single set of connected blocks without empty holes, and (ii) each variable is guaranteed to have been initialised before its value is referenced. Fig. 11 gives an example program.

```
total = 0.0
count = 0
nextStr = input("Number (or end): ")
while nextStr != "end":
    next = float(nextStr)
    total = total + next
    count = count + 1
    nextStr = input("Number (or end): ")
average = total / count
print("The average is", average)
```

Fig. 11. A complete program

V. Discussion and Conclusion

We have presented a design for a blocks-based editor, inspired by visual editors such as Blockly and Scratch, and tailored towards introductory programming in Python. The presented design shares the goals of other blocks-based languages, allowing novice programming to focus on the fundamentals of programming rather than on the accidental complexities of the programming language used. In addition to ensuring syntactically correct programs, the presented design aims to reduce the learner's frustration caused by frequent run-time errors inherent to dynamically typed languages. The design achieves this via declarations of variables' types (within the block palette) which enables types of expressions to be determined during program construction.

The uncomplicated use of color to represent types aims to promote comprehension of the language's type system. We have restricted the discussion to Python's built-in basic types and homogeneous lists. However, the use of color can be readily extended to encompass further types; homogeneous dictionaries can be represented using bi-colored blocks (for the key/value types), and simple class hierarchies might be colored using a single color for the root class, and the same color with various shading patterns added for the subclasses.

Future work will include implementation of the design, and its adaptation and extension to support program execution, function and class definitions, and imports of library modules.

References

[1] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 191–195.

[2] "Blockly," http://code.google.com/p/blockly/, accessed 14 July 2015.

[3] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[4] 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.

[5] 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.

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