# IIA ExA Stimulus Report: Python From Scratch

## Challenges and Potential Solutions for Primary and Secondary Schools in Adapting Text-based Programming Languages from A Block-based Visual Programming Language

Tom Xiaoding Lu
*xl402@cam.ac.uk*
Pembroke College

***Abstract*—This report identifies the challenges faced by children when transitioning to a text-based programming language, having mastered a block-based visual programming language such as *Scratch*, created for educational purposes. Possible solutions to the issues of children preferring debugging via trial-and-error, and difficulties with writing code in perfect syntax are proposed. The report proposes a novice environment which could aid the transition from *Scratch* to *Python*, and discusses the practicalities of such method.**

## I. Introduction

Programming is becoming a more relevant subject for children in primary and secondary level education. In the UK alone, there have been many initiatives introduced to try to train teachers who may be new to programming themselves. The government announced £1.1m of founding for the British Computer Society in 2013, to develop a program for primary school teachers who are new to teaching computing, and a £500,000 fund in February 2014 to attract businesses to help train teachers. Among many initiatives aimed at introducing computing as part of extra-curricular programs in primary and secondary schools, Code Club is by far the most popular, with over 5,000 registered primary schools in the UK participating. Since 2012, over half a million primary school children have been involved in a Code Club, with two million projects created in 2018 [1].

Results published by Smith et al. [1] from surveys of the first year of Code Club in 1000 UK schools, show that Code Club is achieving its objectives, inspiring a sense of fun and achievement for programming and digital creativity. The study also shows that from a sample of 22 'built your own' projects, with no judgment about the presence of bugs, over half of the projects demonstrate tricky concepts such as variables and message passing, along with other important programming concepts shown in figure 1.

In order to aid novices in understanding basic programming concepts without the burden of programming syntax, block-based visual programming languages are normally chosen for educational purposes. *Scratch* is one of the most widely used such programming language adopted by many clubs and classrooms such as Code Clubs. *Scratch* builds on the ideas of Logo [3] but replaces typing code with a drag-and-drop approach inspired by LogoBlocks [4] and EToys [5]. There is a strong emphasis on media manipulation with *Scratch*, it also supports activities that resonate with the interests of youth

| Concept | Correctly used | Incorrectly used |
|---|---|---|
| Variables | 13 | 0 |
| User input | 21 | 1 |
| Broadcast within a sprite | 1 | 0 |
| Broadcasting between sprites | 13 | 2 |
| Control statements | 20 | 1 |
| Boolean connectives | 7 | 0 |
| Parallel execution | 22 | 0 |
| Detecting state | 20 | 0 |

Fig. 1. Concepts used in 'built your own' projects [1]

Fig. 2. User environment offered by *Scratch*

such as creating animated stories and games. An example of its development environment is shown in figure 2

As the final goal for any extra-curricular computer science club is to better prepare children for adapting to production level code taught in higher level education, a question naturally arises regarding the effectiveness of these block-based visual programming languages in leading novices on the way to more powerful programming languages. Meerbaum-Salant et al. [2] have researched how students who have previously learned computer science concepts through *Scratch* faired in comparison to students who had not, when learning production-level languages such as *Java* and C#. The quantitative results, after the first six tests, indicated that the only concept with significant difference between the two groups was for repeated execution in favor of those who had tried *Scratch*. It seems that there exists a bottleneck between programming using a visual programming language designed for novices, and a text-based language which can be used in production. This could potentially result in a lot of novices who do not know what to do after mastering a novice friendly language.

This report aims to review the current literature in the following areas:

- How a production level language such as *Python*, differs from *Scratch*
- Difficulties in learning *Python* for children who are already familiar with basic programming concepts
- Resources available for teachers to help their students transition from *Scratch* to *Python*
- Novel solutions which are designed to aid such transitions
- Suitable evaluation method for evaluating the effectiveness of such solutions

These literature reviews aim to answer one fundamental question: what defines a general purpose programming language or environment which can be easily learned in classrooms after knowing *Scratch*? The report is split into six sections, section II discusses the inspiration of the project, section III is a summary of the literature found in comparing *Python* with *Scratch*. Section IV addresses challenges for both students and teachers who wish to transition the classroom to a text-based programming language, and section V proposes possible solutions and evaluation framework for teachers to provide feedback on the effectiveness of these solutions. Finally conclusion is drawn in section VI on the practicality of the proposed solutions.

## II. Project Setting

Over the course of two academic terms, a classroom with 14 children participating in Code Club were observed. Prior to the session, all children were taught to be proficient in *Scratch*, most of their projects demonstrated complex programming concepts such as event broadcasting and parallel executions. During the first session, the students were introduced to trinket.io, an online *Python* environment used for executing *Python* commands. Much like most of the rudimentary IDEs, trinket provides very limited console debugging output, all of the code is executed through a text editor with syntax highlighting.

Over the course of four consecutive sessions, students were tasked to complete the Astro Pi: Mission Zero challenge [6], the goal of the challenge is to display a coloured message via controlling a raspberry-pi emulator written in *Python*. The students were given detailed instructions and necessary help with the *Python* syntax, they were also introduced to formal ideas of control statements and repeat loops. After all students were able to upload a bug-free version of the control sequences online, students were free to choose either learning *Python* through trinket projects, or returning back to coding in *Scratch*. Figure 4 shows the proportion of students choosing *Python* during all 12 sessions.

The motivation behind this report is due to the sharp drop off observed in figure 4, after students were given a choice between *Python* and *Scratch*. Although this can be explained largely by students' reluctance to code in an unfamiliar language, the most frequently asked questions and coding style demonstrated by students demonstrate that, there were some clear difficulties when transitioning from *Scratch* to *Python*.

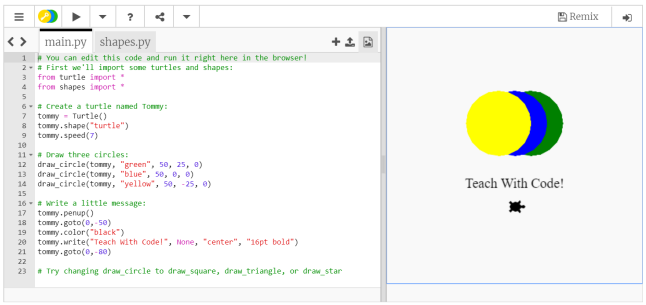



Fig. 3. Example trinket console

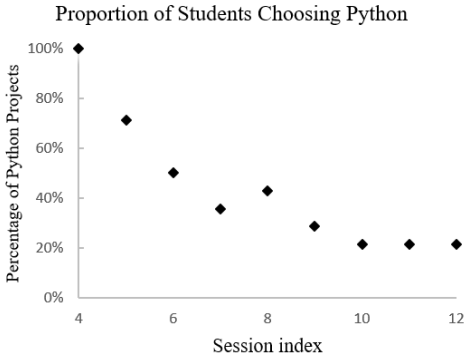



Fig. 4. Proportion of students choosing *Python*

This observation was surprising, since *Scratch* was shown by many papers to be an effective, if not the easiest way for novices who wish to progress to production level coding, the intrinsic understanding of programming concepts was indeed demonstrated by the complex projects they created prior to being introduced *Python*. These observations motivate the discussions and proposals layed out in this report.

## III. *Python* and *Scratch* Discrepancies

*Python* is a descendant of the abc programming language, a language frequently implemented as an alternative to BASIC for teaching programming [7]. The target audience however, has always been professional programmers from the beginning. The usefulness of *Python* as a teaching tool arose later, its educational merits stem from the language's features which promote clean code and simplicity, which are two important traits in a production level language. The pedagogical perspective of the language is captured by its creators in "The Zen of *Python*", with phrases such as "Simple is better than complex" and "If the implementation is hard to explain, it's a bad idea". Formally, *Python* is an imperative language in the same sense as C++, however it is weakly typed and it is the only one which does not use brackets to indicate scope. Due to all these reasons, *Python* is favoured among researchers and data scientists.

To children at the age of primary and secondary schools these advantages are hard to see. The status of *Python* as a production language, on the other hand, is a useful sales pitch. To the question "Why should we learn that" a forthcoming

answer was "because companies like Google and Facebook use it to build the exact thing you see on your computer" [13].
Like all other programming languages aimed at professionals, *Python* has some important traits that are favoured by professionals but increase the learning curve for novices:

- Almost all Integrated Development Environments (IDE) are made with a minimalist approach, as many professionals favour text editors over IDEs
- *Python* on its own does not contain many packages, libraries need to be downloaded separately if any Graphical User Interfaces (GUI) were to be implemented
- Contains specific commands such as def(), ==, ! = which are made to be short at the expense of readability for novices
- Does not provide methods to view the state of each variables unless specified by the programmer

Indeed, all the points above are hugely favoured by professionals, who would much prefer a language which minimizes the code length and storage space on local computers. These traits however, lengthen the learning time for complete novices, often for students who come from a *Scratch* programming background, the minimalist environment seems less intuitive and engaging.
*Scratch* on the other hand, was created with the sole purpose of teaching novices. Programming is done by dragging command blocks from a palette into the scripting pane and assembling them, like puzzle pieces, to create "stacks" of blocks. An individual block or a stack of blocks can be run by double-clicking on it. Multiple stacks can run at the same time so, without realizing it, most *Scratch* users make use of multiple threads [8].
While detailed challenges of learning *Python* are discussed in section IV, one immediate issue *Python* faces is the lack of visual feedback. In *Scratch*, the majority of the screen is occupied by a stage, which the programmer can interact with. *Python* on the other hand lacks any inherent user interactive environment, with at most several buttons for executing the program, uploading files and getting help. Again, as *Python*'s main purpose for professionals is implementing algorithms, due to its Just-In-Time compilation method, it is rarely used for producing GUI applications.

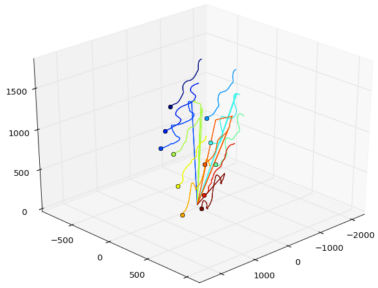
## IV. Challenges

Unlike other disciplines, incorporating programming as part of the primary and secondary curriculum require unconventional framework and resources, these have shown difficulties in their implementation across different education levels and countries. The lack of subject knowledge has led many schools to provide coding lessons, which require students to read through a set of on-screen instructions provided by external parties. This leads to students enthusiastically copying and pasting online instructions and executing them without learning the theory. However, if the conversation is steered towards how their program works, or the concepts of computer science they have used, pupils are often less engaged by teachers who lack confidence in explaining the theory in an engaging yet precise way.
To understand the challenges in teaching computer science at a primary to secondary school level, both challenges facing students and teachers need to be addressed. Section IV-A discusses challenges in learning programming based on children psychology, section IV-B addresses issues on the lack of available teacher's training and resources, which in turn translates to poor content delivered to the students.

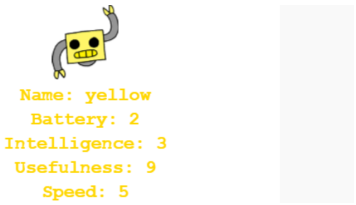
### A. Inherent Challenges for Students

There have been drastic changes over the past decades on the frame work of teaching computer science in both secondary and higher level education. New generations of children are already familiar with computers at the beginning of their primary school education. Although the current generation have natural advantages in both the availability and accessibility of programming materials, programming courses in universities still have on average, the highest dropout rates and are often regarded as the most difficult subject [9]. There is still no understanding why some students learn to program easily and quickly while others have certain difficulties adopting this kind of knowledge [10]. Many research in primary education have been focusing on the cognitive development of children, and how computer science education should be tailored towards their needs. Piaget's theory [11] of cognitive development has shown that children at the age of receiving primary education are entering the concrete operational state of cognitive development. It is characterized by development of logical thinking and problem solving skills. One of the difficulties regarding this development is that children at this age have not yet developed strong abstract, hypothetical thinking, which plays a key role in Computer Science courses. The study also shows that children can only solve problems associated with the real world or something they can relate to. Programming implicitly requires a high level of abstraction and is thus hard to introduce to children to develop an approach that would encourage the development of their logical thinking and problem solving.

*Scratch* is shown to be an effective teaching method as it is a programming language learning environment created with the idea to enable beginners to quickly create programs without having to learn how to write them syntactically correct. This objective resonates with the results shown by Prensky [12] that children are able to learn while playing games, both knowingly and unknowingly. Also, concepts such as the modular design of algorithms can be taught without a lengthy introduction to syntactical details, students can start with simple commands that directly induce a visual feedback, making debugging less time consuming and more engaging.
When children are transitioning to a text-based programming language such as *Python*, the innate nature of this professional facing language makes it less game-like. In order to successfully implement and execute a project, students are forced to develop a strong sense of project abstraction from




(a) A random walk model visualized with matplotlib




(b) Modern Art project output provided by trinket

Fig. 5. *Python* visual display

(a) Dodgeball game using *Scratch*

Fig. 6. *Scratch* visual display

the start, this greatly reduces the complexity of the project student wishes to implement. In *Scratch*, students are able to initially experiment with a grand design and later make simple modifications to the existing framework in order to make the project realizable. In *Python* however, an overly ambitious project cannot be easily salvaged and project definition must be precise both in the flow of operations and details such as the naming of variables. The problem of students unable to make their ideas realizable is aggravated by the lack of visual outputs, as *Python* inherently does not provide any packages that either animates or displays images. Libraries that support these features are either made specifically for scientific purposes (example shown in figure 5(a)) or provide very basic graphics that look underwhelming compared to *Scratch*. An example is shown in figure 5(b) and 6(a), it demonstrates the visual output of one of the most complicated projects found in Code Club *Python* teaching projects, the level of programming required for this project is beyond the level of primary school children (involving classes, dictionaries, etc), yet its visual output is comparable with a basic project implemented in *Scratch*.

The last challenge centers around debugging and syntax precision, these are the two most important factors why children find *Python* frustrating and less game like. In *Scratch*, debugging can be done visually and intuitively, as all control sequences are ”physical” blocks which have defined complementary shapes which reduces error. The physical location in space of these blocks do not matter and there is minimum amount of typing required for even the most complicated projects. *Python* on the other hand, require perfect syntax in order to execute; even undergraduates are often frustrated with its indent sensitivity when first learning *Python*. As all control sequences need to be typed out, with correct indents and brackets, the chance of a student making random errors greatly increase. Again the problem is worsened due to the lack of visual feedback, as students can no longer debug from the animated visual output, but are required to either track the states of different variables or to print them out one by one, both requiring a high amount of attention and logical reasoning. The study done by Konidari et al. [13] concluded that although the intricacies of syntax is a part of programming, it is largely irrelevant for children who have little patience for parsing and language design. *Python* has a simple syntax compared to other languages, but it does place requirements that strike children as arbitrary, like asymmetric ranges and indentation. In this way, it takes a toll on their cognitive burden that it would rather be spent on thinking about problem solving. As syntax is part of programming children must learn to appreciate in order to transition to text-base programming languages, a possible solution discussed in V-B aims to aid children learning the syntax through a *Scratch* like environment, without forcing them explicitly to either memorize or type the correct syntax.

The final challenge in teaching children text-based programming languages arises from the issue with precise syntax requirement. Children’s preferred debugging method is often trail and error, which works well in *Scratch*. The trail and error approach is not only adopted by children, but many adults novice to programming. This method should very much be discouraged as random trial and error in text-based programming languages often do not work and could result in more bugs in the code. One of the key part of learning programming is to be able to solve problems systematically, which is largely demonstrated as one’s ability to debug a program. Section V-C discusses a possible solution at the expense of reduced coding time and higher requirement for teacher’s knowledge in programming.

### B. Challenges for Teachers

Computing is being introduced as a new subject in the school curriculum in many countries, the change in curriculum poses many challenges for teachers who have not had any prior experience in programming. Delivering quality content for Computing classes involves a change to teacher’s practice in both their subject knowledge and pedagogical knowledge, research done by Thompson et al. [15] shows that the lack of confidence was the main reason given by teachers, who could not adopt standards on par with the change in curriculum requirement. Teachers within a context of change face

many challenges, the work done by Finger and Houguet [16] identifies both intrinsic and extrinsic challenges for teachers adapting to the new Computing curriculum which provides the foundation for this section.

Teachers are likely to encounter the challenges that have been identified, or other challenges at any stage throughout the implementation process. These challenges can be broken into two categories: intrinsic challenges involve professional knowledge and understanding of the subject, extrinsic challenges involve resources and professional development. Finger and Houguet [16] identified the following intrinsic challenges which are relevant to this report:

- Professional knowledge and understanding
- Teaching approaches

Similarly, they have also identified the following relevant external challenges:

- A lack of resources
- Practicality of implementation
- Time management
- Methods of evaluating student performance

Both types of challenges reflect the uniqueness of the set of learning goals of the Computing curriculum, and their compatibility issues with Constructive theory, based on the work of Dewey and Piaget [17]. The set of primary learning goals for Computer Science are summarized J. Hromkovic et al. [18] as: introducing the programming language as an example of a formal language, understanding programming as the process of automation and abstraction, and discussing the limits of practical computation power. The concept of a formal language can be acquired by students expanding their vocabulary of programming through experimentation and creating new words of their own, automation and abstraction can be demonstrated through vaguely defining the set up of a project without identifying every detail. Practical computing power i.e. importance of algorithm complexity can be demonstrated with constructing large loops. Many studies aim to decrease the learning difficulty for these three goals under a primary to secondary education setting, making it compatible with the Constructive theory, which emphasizes learning as a cumulative process during which students construct knowledge and meaning for themselves as they learn, and explaining new knowledge in terms of what they already know. Projects such as *Scratch* are successful in achieving the primary learning goals under the framework of Constructive theory, as they enable children to learn as the complexity of their projects increases, without formally spelling out the primary concepts of Computing.

This approach fails, however, when the primary learning goals need to be elevated and formally introduced [9]. When transitioning to a text-based programming language such as *Python*, its learning cannot be achieved without explicitly spelling out the core concepts of abstraction, memory storage, data structures and algorithm complexity. Taking algorithm complexity as an example, students may struggle with looping through what they perceive as a single line of code, yet the code itself is calling a function with high complexity i.e. graphics rendering. This reflects both intrinsic and extrinsic challenges described above, only teachers with a deep understanding in programming can navigate through libraries and explain concepts such as data structure and algorithm complexity, they can then identify issues with children's abstraction early on and intervene before programs written by the student become unsalvageable. Also, the learning of programming syntax is fundamentally incompatible with the Constructive theory, as a program will either execute with fully correct syntax or halt with the smallest syntax error. In *Python*, console error output messages are designed for professionals who are used to the succinct jargon, for children console error messages mean nothing at the beginning of their formal programming lessons. It is very difficult for teachers to deliver engaging content related to syntax correctness. To be able to fully explain reasons behind programming syntax, teachers themselves need to have profound understanding in the subtleties of programming paradigms, otherwise their content will be seen as dull and arbitrary for students.

It is also difficult for teachers to evaluate student performance especially during primary education, it is very difficult to write standardized tests or even finding standardized questions to query students' understanding. Reviewing the code written by students could provide insights to their progress, however, due to reasons discussed previously, much of the code has been copy-and-pasted from project guides, with students getting help from teacher/volunteers whenever syntax error occurs.

In order to tackle these challenges, section V-C proposes a solution which embodies the the "unplugged" approach developed by T. Bell et al. [19] which is aimed at exposing students to ideas from Computer Science without having to use computers (with cards and tapes), while such approaches are used primarily in countries lacking necessary infrastructure to host primary school computing classes, they are also immensely useful in teaching concepts of Computer Science in an engaging way, which student performance can be evaluated in real time.

## V. Possible Solutions

This section aims to propose possible solutions which aid students transitioning to *Python* coming from a *Scratch* background. Of course other programming languages can be adapted, but *Python* is chosen due to its simple syntax and level of support from online communities, it is arguably the fastest production level imperative language to learn. The section is broken down in four parts, section V-A defines in depth, the aspects of transitioning which need to be addressed. Section V-B proposes a block-based programming environment developed to simulate the look and User Interface of *Scratch*, while being made deliberately cumbersome, requiring more and more *Python* syntax as study sessions evolve. Section V-C proposes an engaging method teachers can use to teach students formal concepts of programming, allowing students to develop practical problem abstractions which they can then implement in the environment described above. It is important to note that, projects like *Scratch* inspire students to use their

imagination, creating games while learning programming; Learning production level language decreases the scope of children's imagination, since even when the language itself is mastered, any big projects still require collaboration to be made realizable. Therefore the proposed solutions should not be treated as a substitute for *Scratch*, but as an add-on for those who have mastered *Scratch*.

### A. Problem Formulation

The proposed solutions are aimed towards children with proficiency in *Scratch*, and wish to adapt a text-based programming language such as *Python*. The solutions are developed to be compatible with both Dewey and Piaget's theory [17] of cognitive behaviour and Constructive theory of learning. There are four major problems this section aims to address:

- Is it possible to create a block-based programming environment which is similar to *Scratch*, but teaches students *Python* syntax?
- What are good demonstration projects in this environment which are both engaging and demonstrate real-life usage of production level code such as *Python*?
- Is it possible for teachers to deliver students concepts of formal programming abstractions and control sequences in an engaging way?
- How should teachers evaluate the effectiveness of lessons directly or indirectly?

Section V-B proposes a novel environment which purposefully mimics *Scratch*, with slightly different blocks which enforces the correctness of *Python* syntax, this is then further expanded upon to address the first two points made above. Section V-C proposes a teaching method involving a set of marked color coded cards, to be used at the beginning of each class, aimed at engaging students in discussing concepts of programming abstraction, under a controlled manner. This is then further expanded to an evaluation matrix which teachers can use to evaluate teaching effectiveness, which addresses the last two points made above.

### B. Solution to Syntax Precision

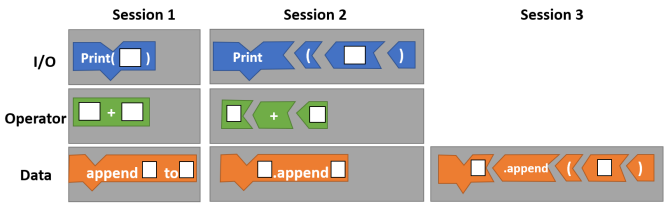



Fig. 7. Block evolution for each session

Difficulties related to syntax precision pose a great challenge for students who are familiar with the drag-and-drop approach offered by *Scratch*. Teaching environments such as trinket require perfect syntax precision even for those who are not yet familiar with *Python* syntax. The ugly text-based interface also seems daunting at first, with a blank page with no available commands which can be viewed. Because of this, the environment offered by trinket is inaccessible by children and appears like a black-box which is only restricted to specific inputs. Children who are initially exposed to the trinket environment waste a lot of time debugging syntax errors such as the usage of brackets and indents, without appreciating the increase in efficiency they offer.

This report proposes a transition environment coded in C++, which has a User Interface strongly resembling one offered by *Scratch* (with a concept shown in figure 8. This introduces a new language with a sense of familiarity, they will still be able to navigate to different command sequences efficiently and using the same drag-and-drop method.

The key difference between this new environment and *Scratch* is the evolution of blocks, instead of verbally descriptive blocks such as "wait until", "when clicked", the new environment uses syntax that resembles *Python* syntax through "block evolution". Figure 7 demonstrates the idea of "block evolution", blocks display syntax that are similar to both *Scratch* and *Python*. For example, with obscure concepts such as the ".append()", during the first session the command is replaced with a more novice friendly version of "a append to b". In later sessions, teacher can choose to evolve the blocks so that they bear stronger resemblance to *Python*. The final evolution should contain all control sequences defined with *Python* syntax, with brackets and operators separated from the rest of control sequences, forcing students to learn heuristics such as "append" should be followed by a pair of brackets. Note as the blocks evolve, simple tasks become more and more cumbersome to perform, as shown in figure 9, to append an item into a list, student will have to drag 5 different blocks to the coding area. This incentivizes students to write their own text-based code, demonstrating the effectiveness of text-based programming languages.

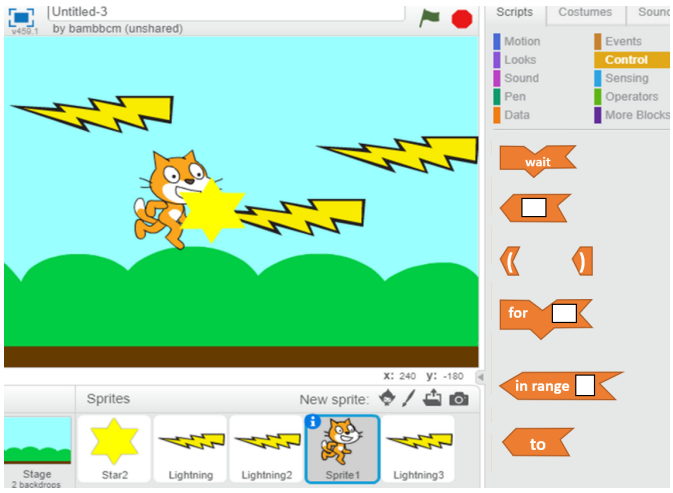



Fig. 8. Ideal User Interface


Fig. 9. Block to text based transition

Through the environment described above, this solution aims to ease the learning curve of mesmerizing syntax, and shows students reasons why text-based programming languages are preferred among the professionals due to the increase in efficiency. Students will also be able to notice that, as blocks evolve, the available number of control sequences become less and less, as blocks such as ”change costume to” will have to be implemented using available control sequences. Writing reusable functions which enable them to create their own ”change costume to” command, demonstrates the modular and customizable feature of programming, some lessons should be focused on abstractions like this, enabling students to rewrite certain functions offered by *Scratch*. Detailed discussions are in section V-C.

One other important discrepancy between this environment and *Scratch* is its emphasis on algorithm implementation. Indeed, children are more easily attracted to being able to change costumes of the spirits, compared with implementing functions which iterate through pixels in order to perform colour transformation. The majority of the *Python* projects aimed at primary and secondary school students however, focus solely on data structure and algorithm implementation, which is both dull and incomprehensible to students at this age (one example project found on Code Club even attempts to demonstrate modulo cryptography). Algorithms are difficult to be implemented in *Scratch*, as it is not designed for this purpose, in the new environment however, as it uses the same syntax as *Python*, algorithms can be easily implemented. More importantly, the algorithms can be run directly on the visual display, providing immediate visual feedback. Good projects should emphasize implementing algorithms that are directly related with the physical world, possible projects include:

- Bouncing ball simulation under gravity
- Minecraft water simulation (destruction of a block results in water pouring out of that block, filling surrounding holes between blocks with water), shown in figure 10
- Trajectory simulation, such as throwing boomerang

Note all possible projects described above share a common feature: fundamentally, they are all maths/logic problems, but their outputs can be mapped onto interactive objects easily on screen. The new environment provides simple commands linking data structures to the behaviour of the objects, while students focus on implementing fundamental algorithms, their output can be visualized easily in a game-like window. This will make the debugging of algorithms more intuitive.

### C. Solution to High Level Abstraction

One observation commonly made by teachers is when learning programming, the majority of students attempt to solve given problems using a trial and error technique. In fact, in most cases, this is the only technique they use, without ever trying to logically analyze a given problem [20]. While this approach may be helpful when debugging block-based programming languages due to the restricted number of combinations of commands, this method is in-feasible when debugging text-based programming language. Many IDEs provide tools for stepping through the code, analyzing memory usage and viewing state of variables, while these debugging tools are powerful, they are often complicated to use and require children to learn a new interface, which adds to the learning curve. It is also worth noting, IDEs are often controversial to use among professionals, with some preferring the simplicity of writing code via a text editor and compiling through terminal. It is therefore vital for children to adapt to debugging through executing code in their head, virtually stepping through the logic and analyzing the problem.

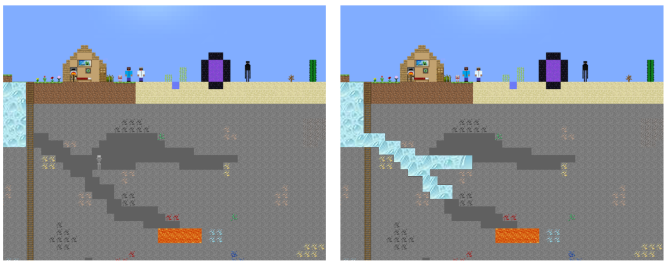


Fig. 10. Reservoir simulation concept

This task is immensely difficult, with many adults failing to visualise the code by just looking at the code. Although programming education in primary and secondary schools is not aimed at producing proficient coders, it is still important for children to transition from a trial-and-error coding approach to an analytical approach. The study done by Tim Bell et al. [19] demonstrates the ”unplugged” style of teaching Computer Science in primary schools. The ”unplugged” style refers to the use of activities to teach Computer Science concepts without the use of computers, and has resulted in many related activities which stimulate an understanding of a concept in a very concrete and practical way. This style of teaching reinforces the level of engagement between students and teachers, at the expense of restricting children’s curiosity. Therefore this style of teaching is suitable after children are ready to transition to a text-based programming language, where sufficient supervision is required due to the higher complexity of the language itself. The formal concept of programming abstraction is difficult to grasp, teachers should spend time engaging students and teach them the correct way of breaking down a task into its constituents.

Basing on the ”unplugged” style with teaching approach similar to method described by Zaharija et al. [20], the following teaching method is proposed. At the beginning of each project created under the new environment described in the previous section, the teacher chooses a project that is

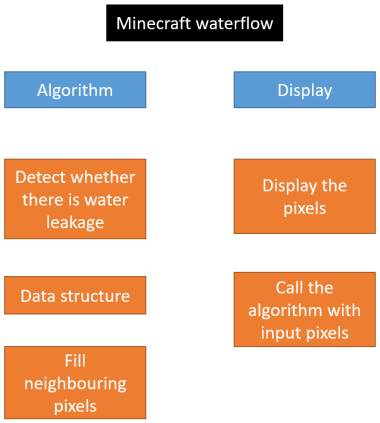



Fig. 11. Reservoir high level abstraction

fully written and is recommended by the community (using the Minecraft reservoir as an example). The teacher will first read through the handout after fully understanding the course, the teacher then makes a set of cards defining each sub-components of the problem (shown in figure 11). In our example, the task of filling pixels where water can pass through is broken down into two parts: algorithm and display, with their own sub-problems. The teacher also prepares some cards that are not relevant to the project. The teacher then leads a group discussion, whereby children are encouraged to discuss possible methods of achieving a project like this, they are then tasked to select cards which are relevant for the project.

After defining the highest level of abstraction of the project, the teacher can then select one key sub-problem as a demonstration, for example, in order to detect whether a square on screen is filled with water, its surrounding four pixels should be checked, if any of them contains water, then the pixel contains water. Concepts like this need to be formalized by students, they are presented with a ”bag of blocks” (shown in figure 12), and are encouraged to obtain what they think, is the logical sequence of steps in order to achieve this. This way, teacher can directly educate students on the formal concept of looping and control sequences, along with usage of functions. After students are equipped with basic concepts of abstraction, and have a clear direction on what to implement, the class can then be split either into groups or individuals, where individual help can be provided.

Note the method described above is not dissimilar to real world software companies brain storming a project, the key difference is that unlike real world where problems may be solved in many ways, here teachers are encouraged to discuss ideas provided by children, but teach students to implement the algorithm in the same way as the handout. Of course, as the competency of students increase, the amount of supervision decreases, aiding independent thinking process.

This solution increases the level of engagement between teacher and students, although formal programming requires high level of supervision, students can still be made to feel that they are steering the classroom.

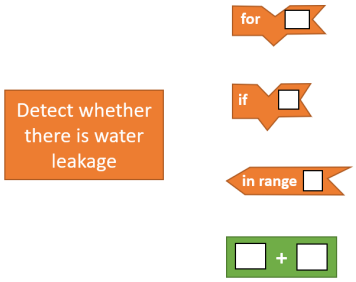



Fig. 12. Block diagram using bag of blocks

### D. Proposed Evaluation Methodology

A key challenge of effective teaching is assessing and monitoring the extent to which students have assimilated the material they were taught. Conventional methods such as questionnaires, tests and essays fail to evaluate children’s ability to code, as there the concept of programming is inherently abstract, and there are many ways to implement even the most basic problem. Concept mapping as a method designed to produce graphical representations of the concepts, viewed from students’ perspective that are important to a given domain and how they are related.

The method of evaluating student performance is as follows:

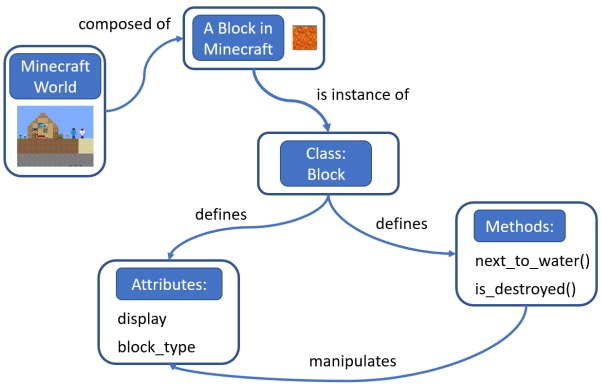



Fig. 13. Teacher’s concept map demonstration

break down key concepts of the programming structure of a project (i.e. class, objects, methods and attributes), transform them into graphical blocks that can be easily understood by Students. The graphical blocks are to be made so that Students can identify the exact parts of code which describe that block. Students are then asked to draw arrows linking the relationships between different blocks, an example is shown in figure 13 (note, this is also the solution to the minecraft reservoir project described earlier). They are encouraged to use certain key words such as ”changes, contains, includes, made up of, is part of, defines, etc”. Their concept maps are then interpreted by the teacher as arrows in the opposite direction but with different descriptions can mean the same thing. These interpreted concept maps are evaluated against teacher’s

concept map, the closeness index (described below) is then calculated, which translate each concept map to a quantifiable metric that is used to evaluate children's understanding of the project.

The closeness index described by Goldsmith et al. describes matric used to evaluate the similarities between a student's concept map and the teacher's concept map. The approach treats all concepts equally important, and focuses on the links between concepts that the two maps have in common. The closeness index of a concept $c$ equals to the number of concepts directly linked to $c$ in both student and teacher's maps, divided by the total number of concepts that are directly linked to $c$ in either maps. Intuitively, if student and teacher have identical number of links to a concept $c$, then closeness index for $c$ is 1; this index drops if the student use either redundant or inadequate number of links to the concept. Formally speaking, the overall closeness index for a student $t$ is denoted as $\mathcal{C}_t$, is calculated using the equation:

$$\mathcal{C}_t = \frac{1}{n}\sum_{i=1}^{n}(S_{i,t} \cap T_i)/(S_{i,t} \cup T_i)$$

Where $n$ denotes the number of concepts that appears in the map, $S_{i,t}$ denotes the set of concepts directly linked to the $i$th of the $n$ concepts in student $t$'s map. The performance metric of the a class of $T$ number of students is therefore the average of their individual concept map closeness scores

$$\mathcal{C}_T = \frac{1}{nT}\sum_{t=1}^{T}\sum_{i=1}^{n}(S_{i,t} \cap T_i)/(S_{i,t} \cup T_i)$$

Note that there are many inherent issues with this metric, it does not take into account which concepts are related to which, it is therefore a first order approximation of the true closeness between students and teacher's concept maps. Also, importance weighting may be added, as some concepts may be more important for children to grasp than others, a modified formula is:

$$\mathcal{C}_{t,\text{weighted}} = \sum_{i=1}^{n} w_i(S_{i,t} \cap T_i)/(S_{i,t} \cup T_i)$$

Subject to

$$\sum_{i}^{n} w_i = 1$$

A reasonable weight to assign to different concepts is treating concepts with more links to be more important, therefore

$$w_i = \frac{T_i}{\sum_i^n T_i}$$

When analyzing qualitatively how students have grasped these concepts, Kinchin and Hay [22] proposed extracting three types of substructure from concept maps: spokes (shown in figure 14), chains (shown in figure 15) and nets (shown in figure 13). Which in essence each corresponds to the hierarchical structure of the concepts learnt. Computing concepts, in nature are organized in a nets substructure. In a spoke substructure the learner has identified certain concepts that are related to a given core, but fails to identify how the former concepts are related to one another. A chain substructure is usually an indication of rote learning, as the sequence depicted by a chain often corresponds to the order in which concepts were introduced in the lecture.

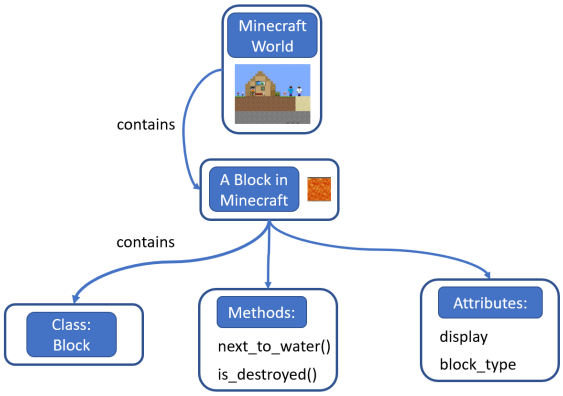



Fig. 14. Spoke structure

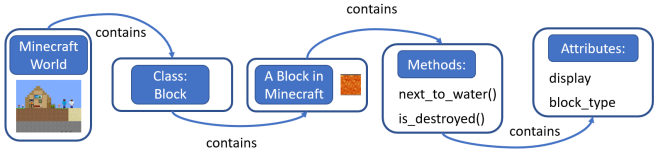



Fig. 15. Chain structure

## VI. Conclusion

Transitioning from *Scratch* to *Python* has many hurdles due to the inherent differences between block-based and text-based programming languages. Some of the key differences include:

- *Python* is professional facing, it has a very basic interface which may seem daunting to novices. *Scratch* is created for novices, with user friendly User Interfaces and visual feedback
- *Python* requires high syntax precision, debugging must be done through logical reasoning. *Scratch* replaces syntax with complementary blocks with descriptions that are made to be informative, trial-and-error debugging approach is feasible
- *Python* is created for implementing complex algorithms and nearly all visualizations are for displaying scientific graphs, *Scratch* is created for users to change the behaviour of characters that were pre-defined
- To implement an entire project in *Python* requires high level abstraction. It requires user to break down the project into its constituents, a technique which is difficult to teach. *Scratch* makes this process easier by allowing user to expand upon example frameworks

- *Python* code can be copy-and-pasted easily, making it difficult for teachers to evaluate the effectiveness of teaching
- Projects created in *Python* are often less visually appealing than *Scratch*. A few changes in *Scratch* results in fancy visual effects, whereas in *Python*, a disproportionate amount of effort is required for even the smallest changes in visual output

It is therefore, difficult to teach children at a primary to secondary level, formal concepts of programming using *Python*, as children struggle with writing code in perfect syntax, their interest in *Python* decreases the more hurdles they need to overcome. This report proposes two solutions to address the problems stated above:

- Create an environment which looks and operates exactly like *Scratch*, but blocks are made more complicated, displaying *Python* syntax. They can then evolve to become more and more like *Python*, at any stage student can choose to abandon drag-and-drop approach, favouring typing their own code
- A new form of teaching framework is proposed, with students being led by the teacher with the aid of coloured cards, to perform high level abstraction and making block diagrams

The report also demonstrates how teachers can effectively evaluate students' performance under the new framework, by performing concept-map matrix evaluation.

The solutions proposed have their shortcomings, they do not address the lack of teacher's resources and training required for teachers to become knowledgeable in the field of programming. Although *Python* has been demonstrated as one of the best programming language for novices, the framework can only be implemented for transition from *Scratch* to *Python*, which has a small target group. These solutions will be ineffective for those who are from a background of Logo or Mindstorm, or those who wish to transition to *Java* or *C++*. Future research could be aiming at fully developing the resources (i.e. software package and handouts) which implement the solutions discussed, and choose a target school as a test point. The solutions described are not effective until it translates to real results in schools or after school programs. There exist clear difficulties for children who are proficient in *Scratch* but wish to transition to *Python*, researches should not only focus on leading children into the field of programming through user-friendly environments, but also equipping children to embark on the logical thinking and design of highly complex real life production level projects.

## REFERENCES

[1] N.Smith C.Sutcliffe L.Sandvik *Code Club: Bringing Programming to UK Primary Schools through Scratch* 2014.
[2] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari *"From Scratch to "real" programming"* ACM Transactions on Computing Education (TOCE), vol.14, no.4, p.25, 2015.4
[3] Papert, S. *Mindstorms*New York: Basic Books, 1980
[4] Begel, A. *LogoBlocks: A Graphical Programming Language for Interacting with the World.* MIT Media Lab, 1996
[5] Resnick, M. Rusk, N. Cooke *Computer Clubhouse: Technological fluency in the inner city.* MA: MIT Press, 1998
[6] https://projects.raspberrypi.org/en/projects/astro-pi-mission-zero
[7] S. Pemberton *An Alternative Simple Language and Environment for PCs* IEEE Software, 4 (1). 56-64.
[8] J. Maloney, K. Peppler, Y.B. Kafai, M. Resnick, N. Rusk *Programming by Choice: Urban Youth Learning Programming with Scratch* 2008 SIGCSE'08
[9] Yadin *Reducing the dropout rate in an introductory programming course* 2011
[10] Wiedenbeck, LaBelle and Kain *PPIG 2004 Factors Affecting Course Outcomes in Introductory Programming* 2004
[11] Wadsworth and Gray *Piaget's theory of cognitive and affective development* 2004
[12] M. Prensky *The Role of Technology in Teaching and Classroom* 2008
[13] Eleni Konidari, Panos Louridas *Why Students Are Not Programmers* 2010
[14] S. Sentance, A. Csizmadia *Computing in the curriculum: Challenges and strategies from a teacher's perspective* 2016
[15] D. Thompson et al. *The role of teachers in implementing curriculum changes* 2013
[16] G. Finger, B. Houguet *Insights into the intrinsic and extrinsic challenges for implementing technology education: case studies of Queensland teachers* 2007
[17] J. Piaget *THE PSYCHOLOGY OF INTELLIGENCE* 1950
[18] J, Hromkovic, T. Kohn, D. Komm, and G. Serafini *Combining the Power of Python with the Simplicity of Logo for a Sustainable Computer Science Education* 2016
[19] T. Bell et al. *Computer Science Unplugged: school students doing real computing without computers* 2009
[20] G. Zaharija, S. Mladenovic, I. Boljat *Introducing basic programming concepts to elementary school children* 2014
[21] Goldsmith, T., Johnson, P., & Action, W *Assessing structural knowledge* 1991 Journal of Educational Psychology, 83, 88-96
[22] Kinchin, I., & Hay, D. *How a qualitative approach to concept map analysis can be used to aid learning by illustrating patterns of conceptual development* 2000