Pyrates: A Serious Game Designed to Support the Transition from Block-Based to Text-Based Programming

Matthieu Branthôme

Pyrates: A Serious Game Designed to Support the Transition from Block-Based to Text-Based Programming

Matthieu Branthôme[0000−0002−9795−0760]

Université de Bretagne Occidentale, CREAD - EA 3875, F-29238 Brest, France
matthieu.branthome@univ-brest.fr

Abstract. This paper presents a design-based research which focuses on the design and the evaluation of the *Pyrates* online application. This serious game aims to introduce Python programming language supporting the transition from block-based languages. The layout of *Pyrates*' learning environment is inspired from beneficial features of block-based programming editors. In order to evaluate this design, the application has been tested in eight classrooms with French 10-th grade students ($n = 240$). Self-generated activity traces have been collected ($n = 69,701$) and supplemented by a qualitative online survey. The data analysis shows that some of the design choices conduct to the expected effects. The creation of a "programming memo" (synthesized documentation) allows the discovery of algorithmic notions while offering a reference support for the Python syntax. The ease of copy-paste from this memo limits keyboarding. The integration of a syntax analyzer designed for beginners gives students a high level of autonomy in handling errors. However, other choices have rather deleterious impacts. For instance, the creation of a control panel for program executions proves to be dedicated to a trial-and-error programming approach or to "notional bypassing" strategies.

Keywords: Block-based programming · Text-based programming · Python · Scratch · Serious game · Design-based research · Learning analytics

1 Introduction

Over the years, block programming has become one of the preferential modalities for introducing computer coding to younger children [6]. Research has demonstrated the benefits of this approach over the traditional introduction using text-based languages [3,18,25]. At the same time, text-based programming remains overwhelmingly used in high school and college contexts for more advanced computer science instruction. This is even more true in industry, where languages like Python and Java are ubiquitous [19]. Learners who started programming with blocks may therefore have to switch to text-based programming. How could they be helped in this transition? This is one of the open questions occupying the research field that focuses on introductory programming [24,16,14,26].

A way to assist them is to design intermediate digital environments offering features that support the transition from one coding modality to the other. These bridging environments are intended to be used transitionally before moving to text-based development tools. *Pyrates* online application [20,21] was developed with this objective. It's a serious game [1] which aims at introducing the Python textual language to high school students.

According to Brousseau [8], one of the drivers of learning is feedback from the "learning environment". He defined this learning environment (called *milieu* in French) as the antagonistic system of the learner, the objects (physical, cultural, social, or human) they interacts with. The *Pyrates*' learning environment was designed taking inspiration from block-based programming editors hoping to take advantage of their features.

This contribution focuses on the evaluation of this design. Hence, the addressed research questions are:

- **RQ1**: During classroom testing, do students adopt the designed features? If so, how do they use them?
- **RQ2**: How do students rate these features regarding clarity and utility?

In this paper, the state-of-the-art related to block-to-text transition is first presented (sect. 2). Next, the design of *Pyrates*' learning environment is outlined (sect. 3). Then, the methodology adopted to evaluate this conception is described (sect. 4) and the ensuing results are exposed and discussed (sect. 5). Finally, the conclusion is followed by some perspectives and extensions (sect. 6).

2 State-of-the-art

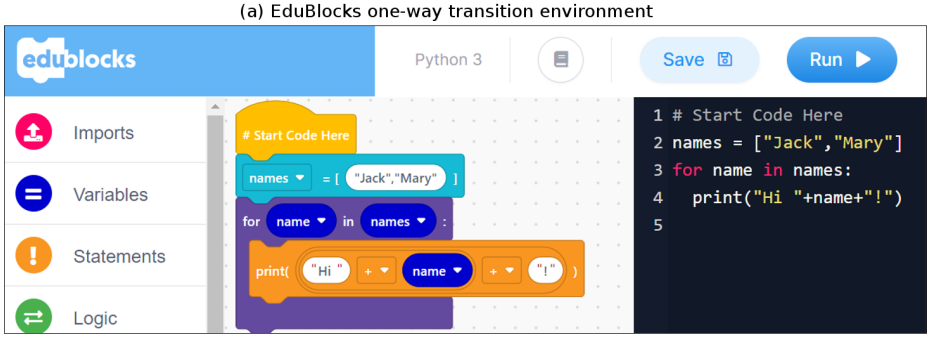
This literature review is divided into two parts. First, existing applications designed to support the transition from blocks to text are presented. Secondly, the results of scientific works analyzing the intrinsic differences between these two kinds of environments are outlined.

2.1 Existing applications

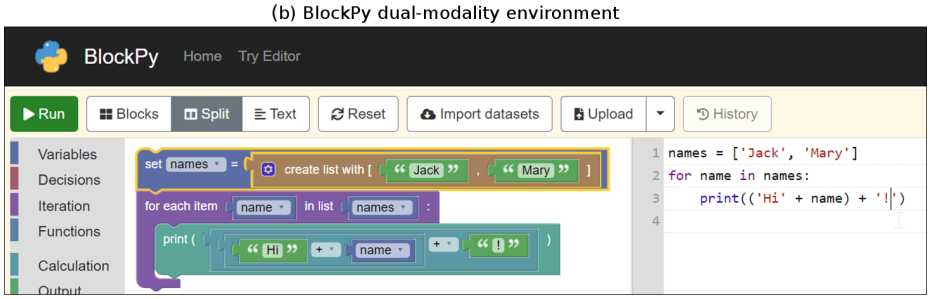
Several avenues based on digital applications have been explored to support the block-to-text transition. Following Lin & Weintrop classification [16], three types of environments are distinguished: one-way transition, dual-modality, and hybrid.

One-way transition environments have two views. One view allows the editing of programs using blocks, these programs being automatically converted into a target textual language in the other view. This target language cannot be directly modified, it can only be consulted and possibly executed by users. This is for example the case of the *EduBlocks* application [10] which automatically translates assembled block-based programs in Python scripts (see Fig. 1-a). The *Patch* environment [23] presents a similar operation based on Scratch blocks.

(a) EduBlocks one-way transition environment



(b) BlockPy dual-modality environment



(c) Strype hybrid environment

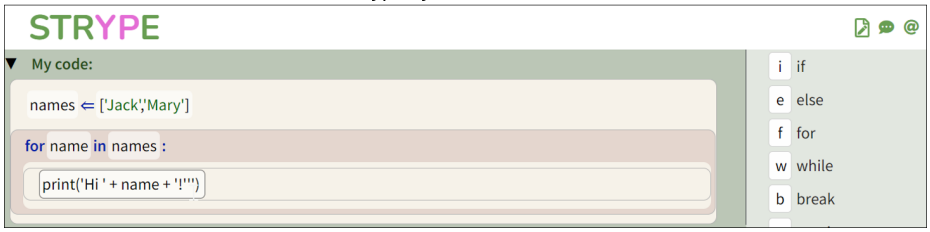


Fig. 1. Examples of the three types of environments

Dual-modality environments are structured in the same manner as one-way ones. In addition, programs can be created or modified directly in the textual view. This automatically results in updating the program in the block view. Existing implementations include *PencilCode* [5], which is aimed at learning Javascript and more recently Python [2]. *BlockPy* [4] provides another environment dedicated to Python programming (see Fig. 1-b).

Finally, **hybrid environments** are combining blocks and text in a single view. High-level structures (loops, conditionals, etc.) can be inserted by drag-and-drop or from keyboard shortcuts. Expression-level code is introduced by traditional text editing supported by auto-completion. *Stride* provides teachers with an operational implementation for the Java language [14]. The freshly released *Strype* [15] offers a "frame-based" environment dedicated to Python edition (see Fig. 1-c).

With respect to this classification, there are actually two types of environments: those based on translation (one-way transition and dual-modality) and those based on the fusion of modalities. Each of them has different objectives. On the one hand, to support the transition on the syntactic and concepts transposition aspects. On the other hand, to temporarily hide the drawbacks of textual languages while still benefiting from the advantages of blocks.

2.2 Advantages of block-based environments: synthesis of the research literature

Several authors [6,14,24] have analyzed the inherent differences between block-based and text-based programming environments. Their most salient results are summarized below.

Availability of a Command Catalog (ACC). Block programming environments present the user with a browsable "palette" listing all existing blocks organized thematically or conceptually. This allows novice users to discover new concepts or to recall previously acquired ones. In text-based environments, the existence and syntax of code structures must be well-known to programmers.

Reduced number of Significant Elements (RSE). Textual programming languages are made up of many units of information (keywords, typographical signs, etc.). This dense notation is an obstacle for novices because it can overwhelm their working memory. Experienced programmers have learned over time to interpret code in larger chunks. Blocks help to reduce the cognitive load of beginners by showing them how to apprehend commands in wider parts.

Drag and Drop Composition (DDC). Composing programs by dragging and dropping blocks limits the difficulty of typing and searching for typographical signs on the keyboard. The purely mechanical act of typing the program text can be a cognitive and motor obstacle for young learners. The need of keyboarding adds cognitive distractions when correcting the inevitable typing errors.

Absence of Syntactic Errors (ASE). Block-based systems avoid most of the syntax errors thanks to a global and constrained manipulation of the structures. In text-based systems, these errors are numerous and the error messages are generally unclear in their formulation. Interpreting these messages is a far from trivial skill which takes a long time for novices to master.

Execution control and visibility (ECV). Block-based environments ease control and improve visibility of program execution. They allow to highlight the block being executed in order to make visible the correspondence between code and action. They may provide a step-by-step mode (set speed, stop and resume execution) or make apparent the current state of variables. These features, not necessarily found in text-based environments, offer to beginners a better understanding of programs execution.

The above comparisons are based on basic code editors. However, some educational text-based environments, like *PyScripter* [22], offer facilitating features such as syntax highlighting, automatic completion, or syntax checking during typing which can help to reduce semantic errors and to limit keyboard input.

3 Design of the learning environment

This section reports on *Pyrates*' learning environnement design. The presentation is based on Fig. 2 which shows the graphical interface and the different areas of the application.

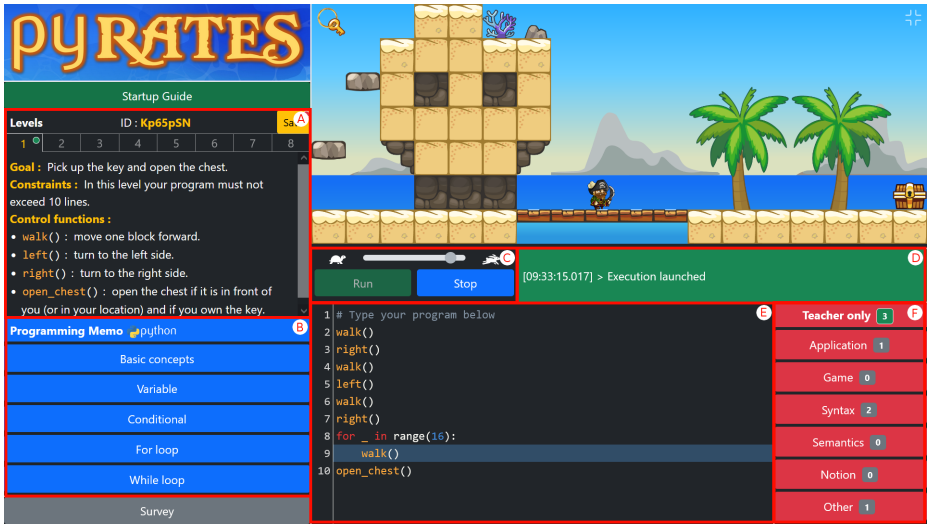


Fig. 2. Different areas of *Pyrates*' graphical interface.

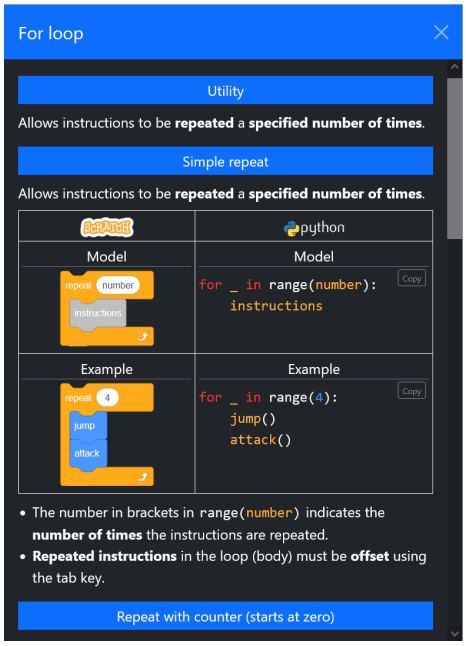
This online application consists of a platform game allowing to control a character using a Python program. This avatar must accomplish various playful objectives. The different levels of this game were designed by implementing the constructivist paradigm which is based on Piaget's psychological hypothesis about adaptive learning [17]. In this way, the algorithmic notions at stake in each level are not explicit but are made necessary by the game problem to be solved. Brousseau [9] qualified these kind of learning situations as "adidactical situations". For the sake of brevity, the game levels' design will not be studied in this paper.

The conception of *Pyrates*' learning environment is presented below. It was designed grounding on the research findings described in section 2.2. Therefore, the features of block programming environments (**ACC** to **ECV**) have been incorporated hoping to take advantage of their benefits.

First, a fixed sidebar was created on the left side of the screen containing, among other elements, a **programming memo** (see Fig. 2-b). This area is inspired by the command catalog present in block-based environments (**ACC**). The memo contents are classified by concepts (basic concepts, variable, conditional, for loop, and while loop) and are accessible by clicking on the different blue buttons. The exposed concepts have been chosen in coherence with the

French mathematics and computer science curriculum. In an effort to guide the students in the exploration of this memo, mouse hovering on a button changes its title by giving an idea of the usefulness of the notion. For example, “variable” becomes “Store information in memory”.

(a) Side panel extract about the « For loop » notion



(b) Side panel extract about the « Conditional» notion

Fig. 3. Two extracts of the programming memo side panel

Clicking on a button causes a side panel appearance detailing the concept in sub-notions (see Fig. 3). Each sub-notion is explained and then illustrated by a **translated generic model and example**. These two programs are expressed both in Python and Scratch languages. Indeed, in France, programming is mainly introduced at lower secondary school using the Scratch block-based language. In this transitional context, Scratch translations of these text-programs are provided. The presence of the Python generic model and its Scratch equivalent is intended to help the learners reducing the number of significant text elements. The goal is to foster the apprehension of Python programs in chunks and not element by element (**RSE**). For example, in the simple repetition case (see Fig. 3-a), students should focus on the number in brackets and consider the rest of the code as a single aggregate.

To limit keyboarding, each piece of Python code is accompanied by a **copy button**. The goal is to encourage the practice of copy-paste to the text editor (see

Fig. 2-e). This usage is a kind of substitute to the drag-and-drop characteristic of block-based environments (**DDC**).

Despite these design efforts, it seems presumptuous to consider the disappearance of syntactic errors (**ASE**). Since interpreting error messages is a hindrance for novice programmers, the learning environment has been enhanced with a research-based **syntax analyzer** especially designed for beginners [13]. This module parses the Python code before interpreter execution. It formulates error messages in users' language (only French and English are currently set up) and in a practical register which novices can understand. Moreover, these messages has been marginally amended according to the programming memo terminology. Thus, when a syntax error occurs, an enhanced message is displayed in the console area of the interface (see Fig. 2-d) and the involved code line is red highlighted in the code editing area. An error-free program does not mean that the code is interpretable. Semantic errors (e.g. related to typing) may still appear during interpretation.

Finally, a **control panel** was created (see Fig. 2-c) to improve the supervision of execution (**ECV**). Users can thus launch and stop program execution and adjust its speed using a slider which changes the speed of characters movements by acting on a multiplying factor. This factor is set to 1 (tortoise) at the launch of the game and can go up to 3 (hare). The visualization of the execution (**ECV**) is ensured by the highlighting of the executed line in the code editor area. In this way, the correspondence between code and current action is apparent.

4 Methodology

This section describes the methodology used to evaluate the design choices exposed in the previous section. This methodology relies on field experiments in classrooms. The *Pyrates* software was tested in eight high school classes in France (10th-grade: 14-15 years old). The 240 involved students were Python beginners.

The students used the application during two or three sessions of 55 minutes each, one or two weeks apart, depending on the class. During the first session, the application and its functioning were quickly introduced before letting the students use it independently during the remaining time. The teacher was asked to intervene only on students' request, or when they had been stuck for a long time. When the teacher interacted with a student, they had to report the content of the given help (application, game, syntax, semantics, notion, other) by clicking on buttons in a reserved frame of the application (see Fig. 2-f).

During these sessions, the application traces the interactions of the students with the learning environment: consultation and copy-paste of contents, syntactic and semantic errors, helps brought by the teacher, launched programs, manipulation of the control panel, etc. These activity traces are automatically generated according to the students' behavior and then exported in a standardized *xAPI* format [12]. This data are completed by an online survey filled in by the students at the end of the experiment. The purpose of this survey is to collect their qualitative point of view on the application.

Consequently, this study data set consists of 69,701 activity traces and 224 survey responses (some students were unable to answer for technical reasons). It was analyzed in an automated way by means of Python programs. Data manipulation and processing relied on *Pandas* library, graphs are generated by *Matplotlib* and *Seaborn* libraries. In an open science approach, the data and the code that led to this paper's figures are shared in an online notebook [7].

5 Results and discussion

The choices described in section 3 have been evaluated by analyzing the students activity traces. In this study, the following traces were taken into account: consultations of the memo, copy-paste from the memo to the code editor, errors detected by the syntax analyzer and by the interpreter, syntactic and semantic aids given by the teachers during their interventions, manipulations of the speed cursor, and chosen speed during the programs' execution.

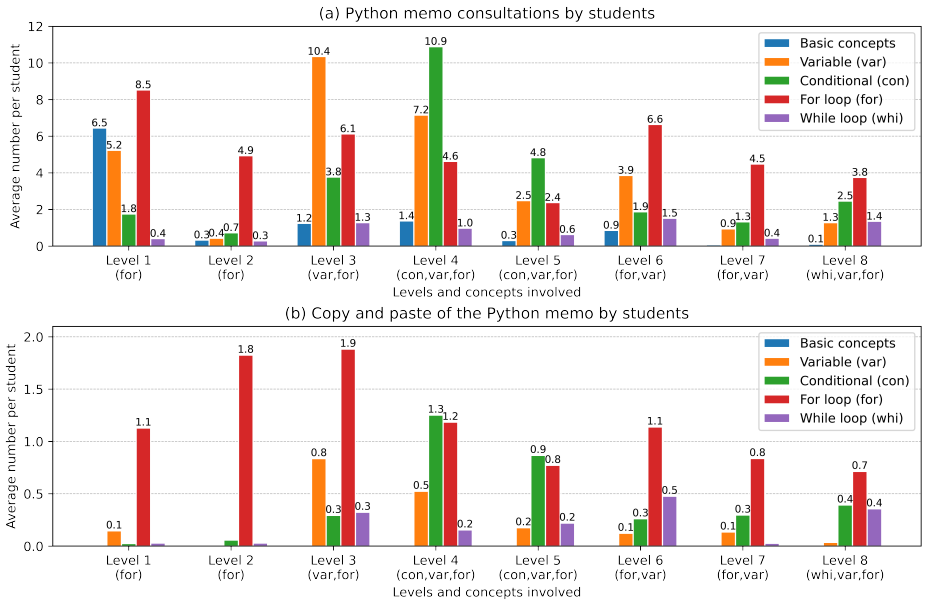


Fig. 4. Consultations and copy-pastes of the Python memo by level

Let us look at programming memo usage. First, Fig. 4-a shows that this memo is frequently consulted by students. It can be noticed that, like the catalog of block-based environments, it supports the discovery of notions. Indeed, each time a new notion is involved in a level (lev 1, lev 3, lev 4, and lev 8), a great variety can be found in the consulted notions. This appears to be the manifestation of a research process. When the concepts have already been used (lev 2, lev 5,

and lev 6), the consultation seems to be more focused on the concepts at stake. The hypothesize can be stated that, in this case, the students need to remember concepts' implementing syntax. This reminds the recall function of the block catalog.

Fig. 4-b allows to assert that the students almost systematically use the copy-paste function when implementing a notion. Each time a notion is involved in a level, there is, on average, at least one use of copy-paste associated with it. Except for the notion of variable which has a much simpler implementation syntax than the other notions. This practice is similar to the drag-and-drop of blocks, and is able to limit keyboard input and help establish code structures.

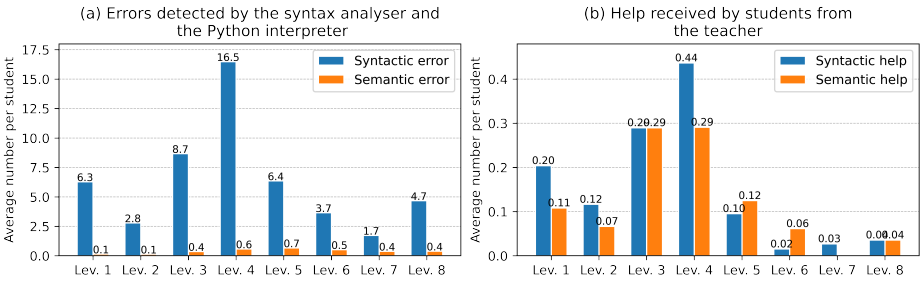


Fig. 5. Errors detected by the application and teacher helps received by students.

Considering errors analysis, the examination of Fig. 5-a shows that syntactic errors (issued from the syntax analyzer) are numerous and in a much higher proportion than semantics ones (issued from the interpreter). Looking at the aids provided by the teachers (see Fig. 5-b), it is remarkable to note that the interventions related to the syntax are very rare. Actually, there is one intervention for every thirty to forty syntactic errors in the first four levels. The students are therefore presumably able to adjust their syntax-erroneous code thanks to feedback from the environment, without asking the teacher.

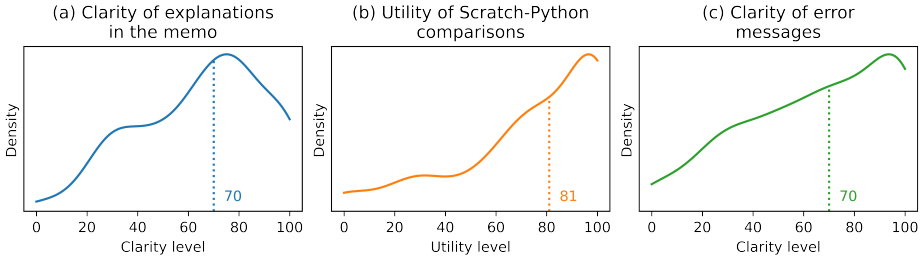


Fig. 6. Results extract from the student survey (score distribution and median).

The traces generated by the application give quantitative insight concerning the use of the memo and the occurrences of the error messages. To go further, these analyses can be qualitatively completed by the survey results. The students had to evaluate several aspects of the application by placing cursors between two extremes ("Not clear" - "Very clear", "Not useful" - "Very useful"), which had the effect of generating a score between 0 and 100. The survey included questions related to the Python memo and the error messages. Fig. 6 presents the scores distribution (density) and median for the these questions.

In addition to being extensively consulted by students, the memo's explanations are considered as clear by the majority of them (see Fig. 6-a). Despite this, a group of students can be distinguished around the score of 30 for whom these contents are more confusing. The comparisons with Scratch are judged as useful or even very useful by the great majority of the students (see Fig. 6-b). Finally, the error messages, which we have shown to foster to students' autonomy, are also deemed to be clear by the largest number of respondents.

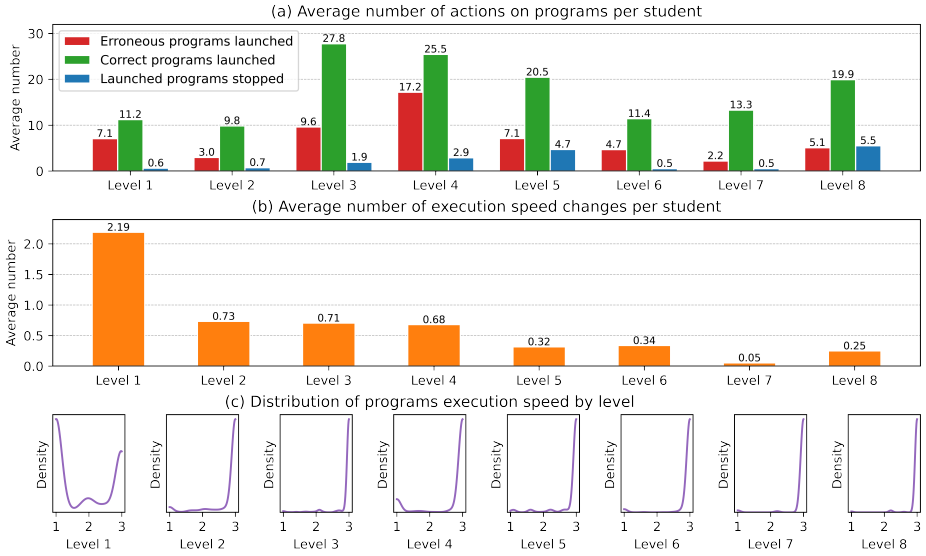


Fig. 7. Data concerning the execution control by level.

Let us now evaluate the use of the program control features. According to Fig. 7-a, there is a very large number of programs run on average per student. Many of them are erroneous, suggesting that students are adopting a trial-and-error programming approach.Numerous correct programs are also launched, which shows that students progress through the game levels in incremental intermediary steps. Program stops are scarce. It is possible to distinguish two types of behaviors depending on the way the levels routes are generated. For a first set of levels with fixed non-random routes (Lev.1, Lev.2, Lev.6, and Lev.7),

students use on average between fifteen and twenty launches and almost no stops. In levels containing random-based routes which change with each run (Lev.3, Lev.4, Lev.5, and Lev.8), students tend to use more launches and to stop some of them. For these random-levels some students adopt a transient operating mode consisting of a series of launch-stop actions until they obtain a random route configuration suitable for their program. This strategy, which can be coined as "notional bypassing", makes it possible to succeed at these levels without implementing the algorithmic notions at stake. These notions are the coding structures based on tests (conditional and while loop). This procedure has very little chance of success because of the large number of different level random routes. These students who remain at any costs in the playful domain are unwilling or unable to enter into notional learning by exploring the learning environment seeking a notion that might allow them to complete the level.

Finally, let us pay attention to the speed change cursor. It is on average rarely used and decreasingly over time (see Fig. 7-b). Fig. 7-c shows the distribution (density) of launched programs' execution speeds for each level. From level 2 onwards, the programs are almost all launched at the maximum speed (multiplying factor of 3). The trial-and-error and incremental programming approach earlier described is consistent with this high execution speed. Indeed, three students remarked in the open-ended field of the survey that "the character does not move fast enough". Nevertheless, a marginal practice can be noted in more advanced levels (level 4 and level 5). It consists of returning to slower execution speeds. Observations during the experiments indicate that some students need to follow more easily the executed lines in a step-by-step action mode.

6 Conclusion and perspectives

To conclude this contribution, its main results can be recalled. The *Pyrates*' learning environment has been designed by incorporating block-based environments features that are thought to be beneficial to students. This design was evaluated by analyzing students' activity and answers to an online survey. Some design choices have the following positive consequences:

- the programming memo is very frequently consulted by the students, it is the support of the discovery and the recall of the concepts ;
- the included comparisons with Scratch are considered useful by a large majority of students, they should help the apprehension of Python structures in larger chunks ;
- copy and paste from the programming memo is widely practiced, this has the effect of limiting keyboarding;
- the feedback provided by the syntax analyzer via "clear" error messages makes it possible to correct the programs with very little teacher involvement.

The control panel should allow the students to better understand the execution of the programs. We note, very marginally, a reduction in the speed of the

character in order to follow the executions in a step-by-step fashion. However, in general, it does not produce the expected results:

- the program launch button is frequently used and the speed control slider is very early set to the maximum in order to adopt a trial-and-error programming approach which do not foster reflection ;
- the button allowing to stop the executions is little used, and when it is, it is mostly to try to succeed in some random-based levels using "notional bypassing".

Beyond these results, in comparaison with the applications presented in section 2, it can be stated that *Pyrates* allows to ease the block-to-text transition at the level of syntax and notions transposition (translated generic model and example). The design environment also partially erase the inconveniences of the text modality while profiting from the benefits of the blocks (programming memo, copy button, control panel and syntax analyser). This application therefore offers an intermediate step, a kind of island, allowing a gradual progression from the block bank to the text bank. However, there is still a step to go towards a more classical practice of programming in Python using a text editor and a command line interpreter.

These results must be considered in light of the limitations of the methodology. Since the students were in a naturalistic context, it was difficult to maintain totally similar experimental conditions between different groups, particularly concerning the teacher's activity and the temporal distance between sessions. Moreover, reasoning only on averages allows to identify trends, but masks the disparities of levels and practices between the students observed in classrooms. Lastly, we did not have the opportunity to measure students' actual learning while playing the *Pyrates* game.

Finally, let us mention some perspectives that can extend this work. Edwards [11] argues that beginners in computer science are more successful at learning if they move from a trial-and-error approach to a "reflection-in-action" practice. Therefore, it would be advantageous to modify the execution control possibilities in our application in such a way as to force students to do less action and more reflection. One way could be to limit the number of executions with scores penalties. Furthermore, it would be interesting to exploit activity traces using data mining algorithms in order to highlight different coding strategies used by students. Clustering algorithms could also be used to identify different student profiles.

Acknowledgements This research was funded by *Region Bretagne*, *Université de Bretagne Occidentale* and *IE-CARE* ANR project. We thank students and teachers who participated in the game evaluation sessions.

References

1. Alvarez, J.: Du jeu vidéo au serious game : approches culturelle, pragmatique et formelle. Ph.D. thesis, Université de Toulouse (2007), https://hal.archives-ouvertes.fr/tel-01240683
2. Andrews, E., Bau, D., Blanchard, J.: From droplet to lilypad: Present and future of dual-modality environments. In: 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 1–2 (2021). https://doi.org/10.1109/vl/hcc51201.2021.9576355
3. Armoni, M., Meerbaum-Salant, O., Ben-Ari, M.: From scratch to "real" programming. ACM Transactions on Computing Education **14**(4), 1–15 (feb 2015). https://doi.org/10.1145/2677087
4. Bart, A., Tibau, J., Tilevich, E., Shaffer, C.A., Kafura, D.: Blockpy: An open access data-science environment for introductory programmers. Computer **50**(05), 18–26 (may 2017). https://doi.org/10.1109/mc.2017.132
5. Bau, D., Bau, D.A., Dawson, M., Pickens, C.S.: Pencil code: Block code for a text world. In: Proceedings of the 14th International Conference on Interaction Design and Children. pp. 445–448. IDC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2771839.2771875
6. Bau, D., Gray, J., Kelleher, C., Sheldon, J., Turbak, F.: Learnable programming: Blocks and beyond. Communications of the ACM **60**(6), 72–80 (may 2017). https://doi.org/10.1145/3015455
7. Branthôme, M.: Paper's data visualisation notebook (2022), https://nbviewer.org/url/storage.py-rates.org/EC-TEL/data_visualisation.ipynb
8. Brousseau, G.: Le contrat didactique: le milieu. Recherches en didactique des mathématiques **9**(3), 309–336 (1990)
9. Brousseau, G.: Théorie des situations didactiques. La Pensée sauvage, Grenoble (1998)
10. Edublocks home page, https://app.edublocks.org/, last accessed 14 Apr 2022
11. Edwards, S.H.: Using software testing to move students from trial-and-error to reflection-in-action. In: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education. pp. 26–30. SIGCSE '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/971300.971312
12. Kevan, J.M., Ryan, P.R.: Experience api: Flexible, decentralized and activity-centric data collection. Technology, Knowledge and Learning **21**(1), 143–149 (Apr 2016). https://doi.org/10.1007/s10758-015-9260-x
13. Kohn, T.: Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment. Ph.D. thesis, ETH Zurich, Zürich (2017). https://doi.org/10.3929/ethz-a-010871088
14. Kölling, M., Brown, N.C.C., Altadmri, A.: Frame-based editing: Easing the transition from blocks to text-based programming. In: Proceedings of the Workshop in Primary and Secondary Computing Education. pp. 29–38. WiPSCE '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2818314.2818331
15. Kyfonidis, C., Weill-Tessier, P., Brown, N.: Strype: Frame-based editing tool for programming the micro:bit through python. In: The 16th Workshop in Primary and Secondary Computing Education. pp. 1–2. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3481312.3481324
16. Lin, Y., Weintrop, D.: The landscape of block-based programming: Characteristics of block-based environments and how they support the transition to text-based

programming. Journal of Computer Languages **67**, 1–18 (2021). https://doi.org/10.1016/j.cola.2021.101075
17. Piaget, J.: L'équilibration des structures cognitives. Presse Universitaire de France, Paris (1975)
18. Price, T.W., Barnes, T.: Comparing textual and block interfaces in a novice programming environment. In: Proceedings of the Eleventh Annual International Conference on International Computing Education Research. pp. 91–99. ICER '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2787622.2787712
19. Pypl - popularity of programming language home page, https://pypl.github.io/PYPL.html, last accessed 14 Apr 2022
20. Pyrates home page, https://py-rates.org, last accessed 14 Apr 2022
21. Pyrates pedagogical guide, https://py-rates.org/guide/EN/, last accessed 14 Apr 2022
22. Pyscripter github page, https://github.com/pyscripter/pyscripter, last accessed 14 Apr 2022
23. Robinson, W.: From scratch to patch: Easing the blocks-text transition. In: Proceedings of the 11th Workshop in Primary and Secondary Computing Education. pp. 96–99. WiPSCE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2978249.2978265
24. Weintrop, D.: Block-based programming in computer science education. Communications of the ACM **62**(8), 22–25 (jul 2019). https://doi.org/10.1145/3341221
25. Weintrop, D., Wilensky, U.: Comparing block-based and text-based programming in high school computer science classrooms. ACM Transactions on Computing Education **18**(1), 1–25 (oct 2017). https://doi.org/10.1145/3089799
26. Weintrop, D., Wilensky, U.: Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. Computers & Education **142**, 1–17 (2019). https://doi.org/10.1016/j.compedu.2019.103646